

Computer Science Department

TECHNICAL REPORT

Compilation Techniques for
VLIW Architectures

F. Gasperoni

Technical Report 435

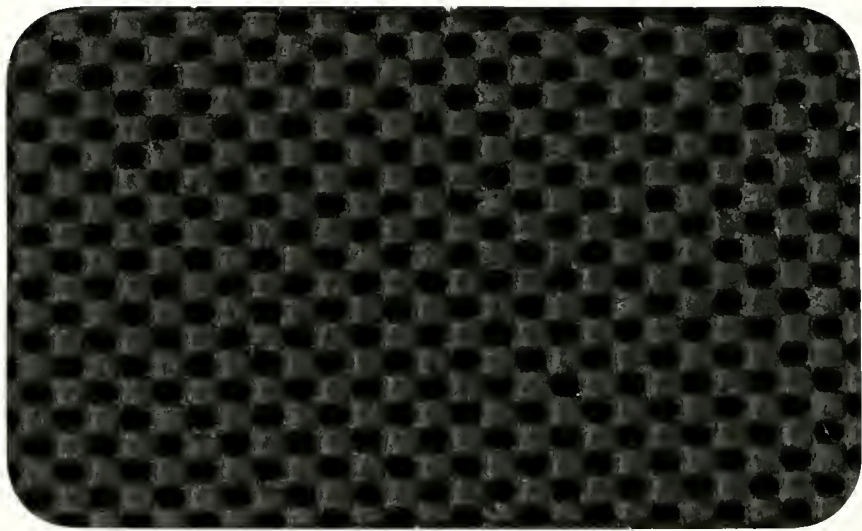
March 1989

NEW YORK UNIVERSITY



Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

NYU COMPSCI TR-435
Gasperoni, Franco
Compilation techniques for
VLIW architectures. c.2



**Compilation Techniques for
VLIW Architectures**

F. Gasperoni

Technical Report 435

March 1989

Compilation Techniques for VLIW Architectures

Franco Gasperoni
gasperon@lang1.cs.nyu.edu
Courant Institute of Mathematical Sciences
New York University

March 1989

Abstract

Experiments have shown that reasonable amounts of fine grain parallelism are available globally in serial programs. New compaction algorithms, as well as a new architecture, have been put forward to exploit such parallelism.

Due to its global nature, ordinary local compaction techniques are inadequate to fully extract the fine grain parallelism available in serial programs. Three global compaction algorithms are presented:

1. *Trace scheduling* compacts the most probable execution paths of a program, using some local compaction algorithm. Compensation code is inserted to preserve semantic equivalence with the original program.
2. *Percolation scheduling* is based on a series of elementary semantic-preserving program transformations which are repeatedly applied using higher level guidance rules.
3. *Region scheduling* evenly redistributes the parallelism available throughout a program, taking into account machine capabilities. Parallelism redistribution is again achieved via repetitive application of elementary program transformations.

Global compaction algorithms alone are insufficient to speedup loops effectively, since after compaction loop's iterations are still executed serially. A new compilation technique, *software pipelining* (the software version of hardware pipelining), overcomes this problem by overlapping successive iterations as much as possible. Several software pipelining algorithms are presented, some achieving optimal or quasi-optimal overlapping in the absence of resource constraints.

The idea behind software pipelining algorithms is to (explicitly or implicitly) determine the (fixed or variable) time interval which must be respected before starting the execution of a new iteration. Most of the algorithms create a rigid pipeline, that is iterations are scheduled when the compiler is certain that they will execute without interruptions. More flexible pipelining algorithms are also presented.

The above compilation techniques have been developed for a specific type of architecture termed very long instruction word (VLIW) architecture. VLIWs vaguely resemble horizontal microengines, but their structure is cleaner and their instruction set more orthogonal.

Three sorts of VLIW machines, based on three different computational models are presented. All three computational paradigms allow multiple operations and multiple conditional branches per machine instruction. What distinguishes the models is the way multiple conditional branches are handled.

The first model was specifically designed to accommodate trace scheduling: semantically multiple conditional jumps are organized like in a Lisp `cond` statement.

The second model, used by percolation scheduling, allows multiple conditional jumps to have the structure of an arbitrary decision tree.

The last model extends percolation scheduling by allowing conditional register assignments to be performed within a decision tree. This model is used to generate flexible software pipelines.

Contents

1	Introduction	1
1.1	Disadvantages of Conventional SIMD/MIMD Architectures	1
1.2	VLIW Architectures	1
1.3	Historical Evolution of VLIW Architectures	3
2	Program Dependence Graph	6
2.1	The Data Dependence Graph	6
2.2	The Control Dependence Graph and the Program Dependence Graph	8
3	Local Compaction	10
3.1	The Local Compaction Problem	10
3.2	List Scheduling	11
4	Global Compaction Techniques	14
4.1	Amount of Fine Grain Parallelism Available in Serial Programs	14
4.2	Trace Scheduling	15
4.2.1	The Trace Scheduling Algorithm	15
4.2.2	VLIW Machines Using Trace Scheduling	17
4.2.3	Problems of Trace Scheduling	21
4.3	Improvements of Trace Scheduling	23
4.3.1	Tree Compaction	23
4.3.2	Singly Rooted Dag Compaction	24
4.4	Percolation Scheduling	26
4.4.1	The Abstract Machine Model	26
4.4.2	The Core and Scheduling Transformations	27
4.4.3	ROPE	30
4.5	Region Scheduling	33
4.5.1	Region Transformations	33
4.5.2	Guidance Rules for Applying Region Transformations	34
4.5.3	From Region Scheduling to Code Generation	36
5	Software Pipelining	37
5.1	Straight Line Software Pipelining Techniques	39
5.1.1	Formal Definition of the Straight Line Software Pipelining Problem	44
5.1.2	A General Algorithm for the Restricted SLSP Problem	47
5.1.3	An Optimal Algorithm for the Restricted SLSP Problem Using Special Hardware	52
5.1.4	An Optimal Algorithm for the General SLSP Problem with Infinite Resources	54
5.2	Complex Software Pipelining Techniques	56
5.2.1	Pipeline Scheduling	56
5.2.2	Perfect Pipelining	64

1 Introduction

1.1 Disadvantages of Conventional SIMD/MIMD Architectures

Multiprocessors and vector machines are the most common types of parallel architectures to date. Porting sequential applications to such architectures involves either a total program rewrite (something the users are reluctant to do) or a recompilation of existing applications with parallelizing compilers.

The first approach provides the biggest opportunity for speedup, since one can design a parallel program suited to the architecture at hand. This is not a painless activity, as it often involves a thorough knowledge of machine idiosyncrasies (for instance when compiling for the Cray-1 the programmer must be aware that vector registers are 64 elements large). Users resort to this approach when the running time of a program is critical.

Acknowledging a user's reluctance to recode working applications, parallelizing compilers try to automatically extract *coarse grain* parallelism from existing (serial) applications. Compilers for multiprocessors [Burke88] must minimize processor communication and synchronization, while devising a fair distribution of the workload (load balancing). This requires the compiler to explore serial programs in search of relatively independent sections of control and data. Compiling for vector machines [Allen87], requires to look for large data aggregates (mainly arrays) which are transformed by simple operations like add and multiply.

Parallelizing compilers are not as effective as programmers in their program transformation task. A parallelizing compiler has to rely on user assertions and/or source code modifications to improve the quality of the code it generates. Furthermore when a user is unable to restructure the source code in a way that the compiler can exploit, part of the code runs serially and machine resources are wasted. This can potentially imply a poor overall speedup. Less user intervention, and better parallel code quality are obtained through thorough program analysis, this includes thorough interprocedural analysis, which is usually hard to obtain.

At any rate, whether manual or automatic parallelization for vector machines or multiprocessors requires matching the *coarse structure* of a program to that of the hardware. This has revealed to be quite a convoluted task.

1.2 VLIW Architectures

Experiments done in the context of numerical programs [Riseman72,Nicolau84b], have shown that vast quantities of *fine grain* parallelism are available in serial applications. This type of parallelism seems easy to exploit in a parallel environment and researchers [Arvind80,Fisher84c,Carnevali86,Ebcioğlu88] have suggested its use over coarse grain parallelism.

There are two distinct schools of thought for the use of fine grain parallelism. One advocates its exploitation at run time (dataflow languages and architectures), the other its automatic extraction at compile time. Compile time extraction of fine grain parallelism is the subject of this survey paper. For a survey of the dataflow approach see [Veen86].

What characterizes fine grain computations is the availability, at any given time, of a good number of simple, but irregular operations acting upon independent data. This suggests an architecture with independent functional units arranged in a way such that compile time scheduling of fine grain operations is possible and entails practically zero run time overhead. The new architecture introduced to meet these demands is

inspired from horizontally microcoded engines. The archetype architecture (see figure 1) comprises a number of integer/floating ALUs, one or several register files, data memory and a central control unit. All machine

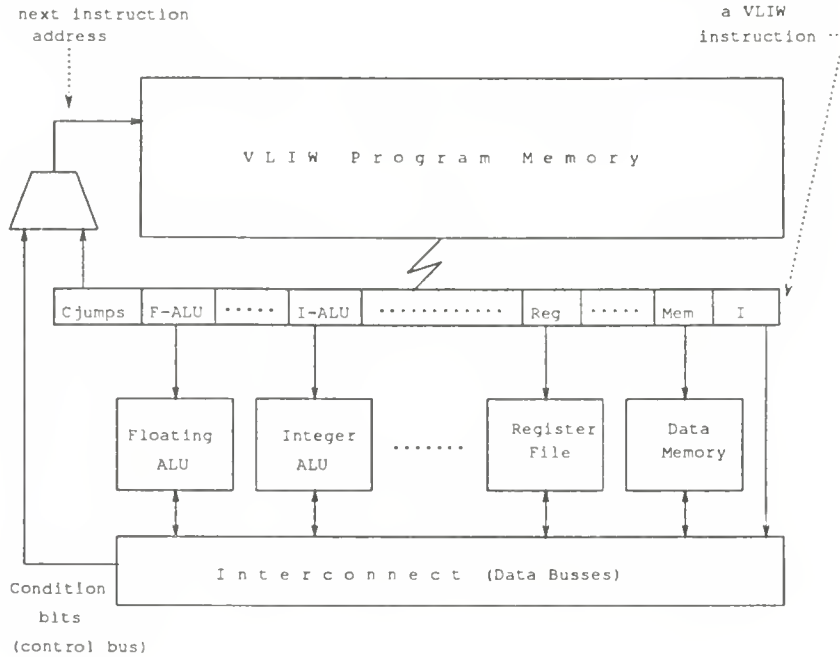


Figure 1: General VLIW architecture.

resources are driven by the same clock, and are connected using some sort of topology. The instructions which operate each individual unit are gathered in a *very long instruction word* whose size can go from 100 to more than 1,000 bits depending on the type and number of machine resources. The architecture has been termed very long instruction word (VLIW) architecture. In first approximation a very long instruction word is similar to a horizontal microcode instruction, except that it is much longer and much more orthogonal. Like in horizontal microcode, VLIW instructions are the last layer in the software hierarchy: there is no additional level of interpretation.

VLIW machines can be assimilated to *synchronous* MIMDs [Ebcioğlu88]. In fact let $\mathcal{M}$ be an ideal shared memory synchronous MIMD machine containing p RISC-like processors executing an instruction every cycle. Suppose the p processors execute p different programs comprising $n_1, \dots, n_p$ instructions respectively. Then $\mathcal{M}$ can be simulated by a VLIW machine with complex conditional branching, that is a machine which can branch to any of 2^p labels given a set of p independent tests. The synchronous MIMD programs can be translated into a VLIW program with at most $n_1 \cdot \dots \cdot n_p$ instructions. Conversely a VLIW machine with complex conditional branching can be trivially simulated by a synchronous MIMD.

The key point of VLIW architectures is the symbiosis between hardware and software: every machine resource is completely and independently controlled by the compiler, that is:

- resources are tightly coupled, there is a single instruction stream initiating a set of fine grain operations every cycle¹,
- every operation executed by a machine resource takes a predictable amount of time; functional units may be pipelined, but there are no hardware interlocks, the compiler has to take into account possible data dependences and conflicts when scheduling program operations,
- all communications between machine units are completely orchestrated by the compiler: the source and destination, as well as the time needed to transfer data between resources are all known to the compiler which, based on this information, can devise an efficient schedule.

The input to a VLIW compiler is some sort of sequential intermediate language, like triples or quadruples, that has undergone extensive conventional optimization [Aho86]. The goal of a VLIW compiler is to schedule every cycle as many fine grain operations as there are fields in a VLIW instruction, in a way such that program semantics are preserved and execution time minimized. The process of combining several intermediate operations into a VLIW instruction is called *compaction*, and bears resemblance with the code compaction process that takes place in horizontally microcoded machines. Code compaction can be one of two kinds:

local compaction: is restricted to basic blocks, that is each VLIW instruction is built from operations of a same basic block,

global compaction: disregards basic block boundaries, VLIW instructions may comprise operations from different basic blocks.

Local compaction is generally inadequate for VLIW machines, as there isn't enough fine grain parallelism to exploit within a basic block [Riseman72].

1.3 Historical Evolution of VLIW Architectures

VLIW architectures evolved from research in horizontally microcoded machines. Horizontal microengines dispose of a small number of non uniform hardware resources connected in an irregular fashion. The advantage of horizontal over vertical microarchitectures is the ability to execute several microoperations concurrently. This performance increase comes at the cost of more complex programming, as several (possibly conflicting) parallel activities have to be specified for each clock cycle. To ease the programming effort of horizontal microengines, microcode compilers have been introduced to avoid hand-coding of such machines: a microcode compiler takes a vertical list of microoperations and compacts them into horizontal microinstructions.

Till the late 70's microcode compaction was essentially local (see [Landskov80] for a survey of local microcode compaction). The local compaction problem (section 3) is NP-complete, but algorithms such as *list scheduling* (section 3.2), inspired from job scheduling theory [Coffman76] very often produce results which are within a few percent from the time optimum. As horizontal microengines became more and more complex, allowing more concurrent operations to be specified every cycle, the need grew to employ more global techniques. In 1978 Tokoro et al. [Tokoro78] introduced one such technique, where local compaction is first applied to each basic block, and then motion of operations between adjacent basic blocks is considered, as

¹From here on the word *instruction* refers to a VLIW instruction, whereas the word *operation* refers to one of the fine grain operations composing a VLIW instruction.

allowed by a list of legal moves. The method remains inherently local as the local compaction phase imposes too many arbitrary constraints which limit the applicability of the global code motion rules. In 1979 Fisher [Fisher79] introduced an interesting global compaction technique, *trace scheduling* (from here on TS) which selects the most probable execution paths of a microprogram and locally compacts such paths, disregarding conditional jumps (compensation code is inserted to preserve semantic equivalence with the original program). It is the adaptation of such technique to high level programs that gave birth to the field of VLIWs.

Experiments done in the early 70s and repeated in the early 80s (section 4.1) suggest that great amounts of fine grain parallelism are available in ordinary programs. In the 70s such parallelism was discarded on the basis of its global nature and the inability of constructing efficient hardware mechanisms to exploit it. Fisher realized that TS could be adapted (section 4.2) to extract a good part of the parallelism globally available in serial programs. To exploit such parallelism Fisher and his team designed the ELI, the first true VLIW architecture. The machine model used in the ELI accommodates the needs of TS, by allowing multiple operations and conditional jumps per VLIW instruction. The conditional jumps are organized like in a lisp `cond` statement. The ELI project gave birth to a company, Multiflow Inc. whose goal was to produce a commercial product based on trace scheduling and VLIW technology. A line of machines called the TRACE(TM) was eventually introduced in 1988.

Despite its good results TS has some problems, the biggest of which is the (potentially exponential) increase in code size caused by the insertion of compensation code. This is a very acute problem in microcoded machines where microprogramming memory is of reduced size and space is at a premium. Methods have been introduced in the microprogramming world (section 4.3) to overcome code explosion and some other problems of TS.

As a result of his experience with TS, Nicolau [Nicolau84a] has introduced another global compaction technique that formalizes the ideas behind TS (section 4.4). The new global compaction technique is based on a series of semantic preserving elementary program transformations, that are repeatedly applied using some high level guidance rules. The repeated application of such elementary transformations allows operations to percolate to the beginning of a program. The machine model that Nicolau uses for his global compaction technique (called *percolation scheduling*) extends the model used for TS, by allowing arbitrary decision trees in each VLIW instruction.

The last global compaction technique presented in this paper is *region scheduling* (section 4.5) developed by Gupta in 1987. Region scheduling works by evenly redistributing the parallelism available throughout a program, taking into account machine capabilities. Like for percolation scheduling parallelism is redistributed through repetitive application of semantic preserving elementary program transformations. The elementary transformations are however not defined in terms of a program's flow graph, but rather a new intermediate program representation paradigm: the *program dependence graph* [Ferrante87]. The program dependence graph is better suited than the program flow graph for the extraction of fine grain parallelism, as no spurious ordering constraints are imposed on program operations.

The previous global compaction techniques are unable to speed up loops significantly since after compaction loop iterations are still executed serially. Once more a technique used in the microprogramming world [Kogge77] has been adapted to VLIWs. Such technique, termed *software pipelining*, overlaps loop

iterations as much as resource and data dependences permit. Simple software pipelining algorithms have been presented as early as 1981 for the FPS-164 (section 5.1) a precursor of VLIW machines. Better software pipelining algorithms for the FPS were given in 1984 and 1986 respectively by Touzeau and Su (section 5.1). However it is not until recently (1987/1988) that more sophisticated loop pipelining algorithms have been introduced. Lam (section 5.1.2) has presented an algorithm that can process arbitrary loops, and produce near time optimal results for loops with no IF statements. Aiken and Nicolau (section 5.1.4) have discovered a polynomial time algorithm, *greedy scheduling* that produces time optimal results for synchronous MIMD machines, for loops without IF statements.

The previous algorithms all assume machines allowing only one conditional jump per VLIW instruction. In 1987 Ebcioğlu has introduced a machine model extending the percolation scheduling model by allowing multiple conditional register assignments within the decision tree of a percolation scheduling instruction. Ebcioğlu employs such machine model in conjunction with a pipelining algorithm called *pipeline scheduling* to pipeline inherently sequential loops (section 5.2.1). In 1988 Aiken and Nicolau have presented another pipelining technique (based on the percolation scheduling model) to generate code for VLIW machines with multiple conditional branches. The technique, called *perfect pipelining* has the same effect as unbounded loop unrolling followed by compaction.

2 Program Dependence Graph

Serial or parallel optimization techniques need to preserve the “characteristics” of the initial program, more precisely given a program $\mathcal{P}$, any semantic-preserving program transformation has to respect the partial ordering of operations imposed by $\mathcal{P}$ ’s *data* and *control* dependences.

Flow graphs [Aho86] are the most common way of representing control dependences. To express data dependences flow graphs are usually supplemented with definition-use and/or use-definition chains [Aho86].

For parallel optimizations *data dependence graphs* [Kuck81] (section 2.1) formalize the notion of data dependence and are a more appropriate representation than definition-use and use-definition chains. Likewise *control dependence graphs* [Ferrante87] (section 2.2) are more appropriate than flow graphs to represent control dependences as they more faithfully reflect a program’s control structure (flow graphs imply an excessive number of control dependences). By combining the control and data dependence graphs one obtains an intermediate program representation, called the *program dependence graph* [Ferrante87], that adequately reflects the essential characteristics of a program.

While data dependence graphs have been used for more than a decade and have now become a standard tool in parallel optimization techniques, control dependence graphs have been introduced only recently. Consequently all of the compaction techniques herein presented, except for region scheduling (section 4.5), are formulated in terms of flow and data dependence graphs.

2.1 The Data Dependence Graph

Let op_1 and op_2 be two operations of a program $\mathcal{P}$, operation op_2 is said to be *data dependent* on operation op_1 if there is a path from op_1 to op_2 in $\mathcal{P}$ ’s flow graph, and one of the two operations modifies a variable that the other operation uses or also modifies. The three possibilities are termed as follows:

- *flow dependence*: op_1 modifies a variable used in op_2 (write-read dependence);
- *anti-dependence*: op_1 uses a variable that op_2 modifies (read-write dependence);
- *output dependence*: both op_1 and op_2 modify the same variable (write-write dependence);

Of the above the only “true” dependence is flow dependence, as it is part of an algorithm data flow and cannot be removed by simple transformations. Anti or output dependences arise because of the programmer concern to reuse storage. Often these false dependences can be removed by copying or renaming [Kuck81,Padua86].

The data dependence graph (DDG) represents $\mathcal{P}$ ’s data dependences. The DDG is a directed multigraph, whose nodes are $\mathcal{P}$ ’s operations (in what follows we identify an operation with the node representing it). There is an edge from op_1 to op_2 , if operation op_2 *may*² depend on op_1 .

If $\mathcal{P}$ ’s flow graph is acyclic the DDG is a dag (data dependence dag or DDD). When dealing with loops some further precisions are needed. Operations contained in loops are executed several times and thus dependences can arise between different instances of the same operation, or between different instances of different operations. This type of dependence is termed *loop carried* (or inter-loop) dependence [Allen87]. Normal data dependences occurring between operations of the same iteration are called *loop independent* (or intra-loop) dependences. In the context of VLIWs we are mainly interested in loop carried dependences

²Even if we are not certain that op_2 will depend on op_1 , we must insert a dependence edge, to insure correct execution. A “may” dependence edge is needed, for instance, between two operations that could potentially modify the same variable.

of innermost loops. Let $\mathcal{L}$ be some innermost loop in $\mathcal{P}$. We assume that $\mathcal{L}$ has been normalized to start iterating at 1, and has a loop increment of 1. Let op_1 and op_2 be two operations occurring in $\mathcal{L}$. If the

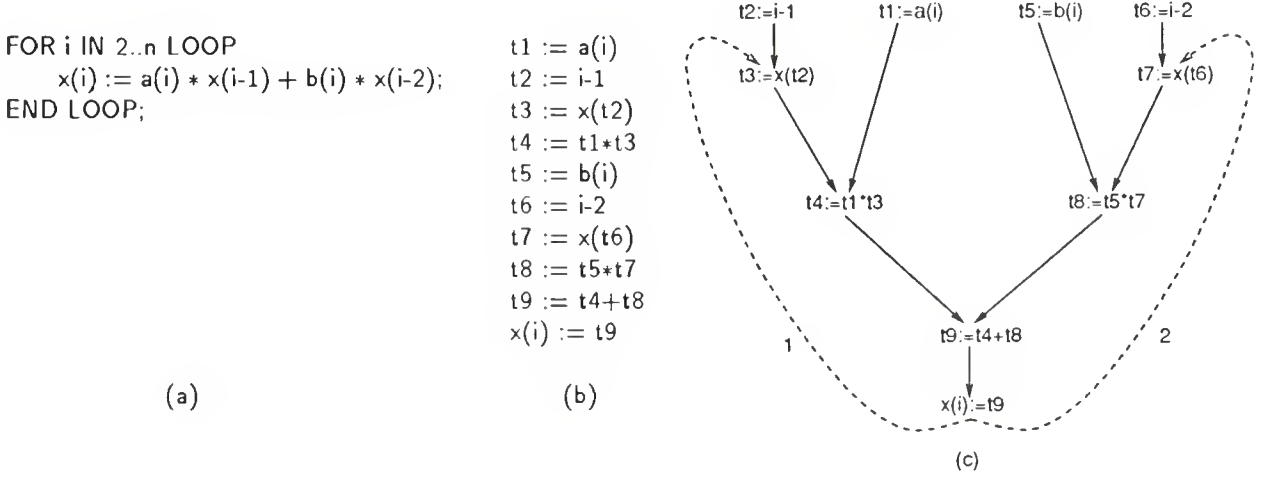


Figure 2: (a) A loop. (b) The intermediate code for the loop body. (c) The loop DDG. Loop carried dependence edges are dashed (loop carried dependences introduced by temporaries are ignored).

execution of operation op_2 , during iteration j may depend on the execution of operation op_1 on iteration i , $i < j$, we create an edge from op_1 to op_2 labeled by the value $j - i$. We call $j - i$ the *lag* of the loop carried dependence. For an edge e we define $\lambda(e)$ to be the minimum lag associated with e when e represents a loop carried dependence, otherwise $\lambda(e) = 0$. Loop carried dependences may introduce cycles in the DDG. This happens, for instance, when values computed during an iteration are used as inputs in subsequent iterations (this type of loop is called a *recurrence*).

As we have previously said, each edge e has attached a value $\lambda(e)$. When dealing with concrete machines we also associate to each edge $e = (op_1, op_2)$ the minimum delay that must be respected between the execution of op_1 and op_2 . We call this value $\delta(e)$. Thus every edge $e = (op_1, op_2)$ has an associated tuple $[\lambda(e), \delta(e)]$, whose meaning is:

- $\lambda(e) = 0$: operation op_2 must execute $\delta(e)$ cycles after the execution of op_1 has begun;
- $\lambda(e) \neq 0$: operation op_2 must execute $\delta(e)$ cycles after the execution of op_1 of the $\lambda(e)$ th previous iteration has begun.

Note that $\delta(op_1, op_2)$ might be less than the total execution time of op_1 , as op_1 might produce a result before its execution completes.

As an example consider the loop in figure 2(a). The intermediate code for the loop body is given in figure 2(b), the loop data dependence dag is given in figure 2(c).

2.2 The Control Dependence Graph and the Program Dependence Graph

Informally an operation op is said to be control dependent on some test t , if op can only be executed as a result of following one of t 's branches. Formally let $\mathcal{P}$ be some program and let $\mathcal{F}$ be its flow graph ($\mathcal{F}$ is assumed to have a unique exit node called $exit$, and vertices to contain a single operation). An operation op_1 of $\mathcal{F}$ is said to be *post-dominated* by an operation op_2 , if every path from op_1 to $exit$ contains op_2 . An operation op of $\mathcal{F}$ is *control dependent* on a test t of $\mathcal{F}$ if and only if:

there exists a path P from t to op , such that any operation in P (except for t) is post-dominated by op , and t is not post-dominated by op .

The *control dependence graph* (CDG) of a program expresses control dependencies in the same manner the DDG expresses data dependencies. CDG vertex set is composed of $\mathcal{P}$'s operations. Let op_1 and op_2 be two such operations. Then there is an edge from op_1 to op_2 in the CDG, if and only if op_2 is control dependent on op_1 (op_1 is a test). Edges are labeled true or false, indicating the condition under which an operation is executed. See figure 3(a) and (b) for an example.

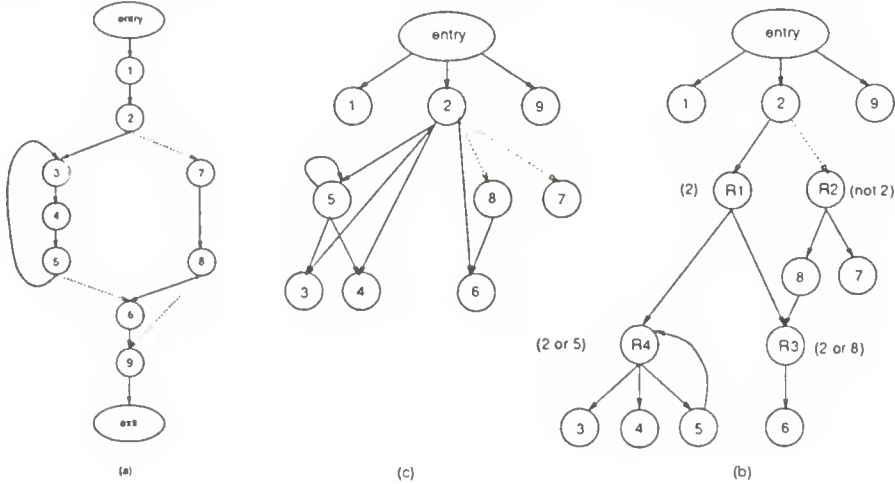


Figure 3: (a) Initial flow graph. Dotted edges indicate branches taken on false conditionals. (b) The corresponding CDG. Edges labeled true are plain, edges labeled false are dotted. (c) The final CDG after region nodes have been added. The numbers next to a region node indicate the predicate which must hold for the region to execute.

To summarize the set of control conditions for an operation, and group all the operations with the same set of control dependencies together, *region nodes* are introduced. In what follows, two operations are said to have same control dependence predecessors if each has control dependence edges from exactly the same nodes and the corresponding edges are labeled with the same true or false labels. Region nodes are created as follows:

1. Consider an operation op which has no region node predecessor in the CDG. Let $Pred$ be the set of control dependence predecessors of op .
2. A region node R is created for $Pred$. Each operation whose set of control dependence predecessors is $Pred$ is made to have the single control dependence predecessor R .
3. R 's set of control dependence predecessors is set to be $Pred$.
4. If there exists a region R' whose set of control dependence predecessors contains $Pred$, R' is made to be directly control dependent on R and the edges corresponding to $Pred$ in R' are deleted.
5. The previous steps are repeated until step 1 no longer applies.

Essentially, a region node represents a conditional expression which, when true, causes the operations dependent on it to execute (in parallel if there are no data dependences). See figure 3(c).

The program dependence graph (PDG) combines the control and data dependence graphs of a program. The PDG vertex set comprises the program's operations and the CDG region nodes, the PDG edge set is the union of the DDG and CDG edge sets. While the PDG explicitly represents the data and control dependences that exist in a program $\mathcal{P}$, $\mathcal{P}$'s flow graph does not have such a desirable property as it imposes a fixed sequencing of operations that need not hold for an algorithmically and semantically equivalent program $\mathcal{P}'$.

3 Local Compaction

In the following sections the *local compaction problem* (LCP) is formally defined and *list scheduling*, a popular heuristic for performing local compaction, is presented. Despite the NP-completeness of LCP, list scheduling produces solutions which are generally within a few percent of the time optimum.

3.1 The Local Compaction Problem

Informally the local compaction problem (LCP) is: given a basic block (that is a set of operations along with their data dependences), and given machine resource constraints, the goal is to schedule the operations in a time-optimal fashion such that resource and data dependence constraints are respected. Let $\mathbf{N}$ be the set of naturals, and $\mathbf{Z}$ be the set of integers. Formally the local compaction problem is: **given**

1. a machine $\mathcal{M}$, a set of resources $\mathcal{R} = \{r_1, \dots, r_m\}$ that the machine possess,
2. a resource configuration vector $\vec{R}_{\mathcal{M}}$ of $\mathbf{N}^m$, where the k th entry of $\vec{R}_{\mathcal{M}}$ (denoted $\vec{R}_{\mathcal{M}}(k)$) gives the number of units of resource r_k available in the machine configuration,
3. a set of l operations $\mathcal{O} = \{op_1, \dots, op_j, \dots, op_l\}$,
4. a duration function $d : \mathcal{O} \rightarrow \mathbf{N}$, where $d(op_j)$ is the number of machine cycles op_j takes to execute,
5. a resource usage function $\vec{R}_{\mathcal{O}} : \mathcal{O} \times \mathbf{Z} \rightarrow \mathbf{N}^m$ ($\vec{0}$ is the null vector):

$$\vec{R}_{\mathcal{O}}(op_j, x) = \begin{cases} \text{the resource usage vector of the } x\text{th step of operation } op_j & \text{if } 0 \leq x < d(op_j) \\ \vec{0} & \text{otherwise} \end{cases}$$

the k th entry of $\vec{R}_{\mathcal{O}}(op_j, x)$ (denoted $\vec{R}_{\mathcal{O}}(op_j, x)(k)$) gives the number of units of resource r_k needed in the x th time step of operation op_j ,

6. a data dependence dag $\text{DDD} = (\mathcal{O}, E)$ imposing a partial ordering on $\mathcal{O}$ (DDD embodies the data dependences of $\mathcal{O}$'s operations),
7. a delay function $\delta : E \rightarrow \mathbf{N}$, defined on the edges of DDD, where for $e = (op_{j_1}, op_{j_2})$, $\delta(e)$ is the delay that has to be respected before scheduling op_{j_2} , once op_{j_1} has been scheduled (this is the delay function of section 2.1).

The **goal** is to find a schedule $\sigma : \mathcal{O} \rightarrow \mathbf{N}$ such that:

1. **minimality:** σ is of minimum length. The length of a schedule σ , $\text{len}(\sigma) = \max_{op \in \mathcal{O}} (\sigma(op) + d(op))$
2. **dependence constraints:** $\forall e = (op_{j_1}, op_{j_2}) \in E$, $\sigma(op_{j_2}) - \sigma(op_{j_1}) \geq \delta(e)$
3. **resource constraints:** define vector addition in the usual way, and vector comparison $\preceq$ to be: $\vec{v}_1 \preceq \vec{v}_2$ iff $\forall 0 \leq k \leq m$ $\vec{v}_1(k) \leq \vec{v}_2(k)$, then the resource constraints are:

$$\forall 0 \leq t \leq \text{len}(\sigma) \quad \sum_{j=1}^l \vec{R}_{\mathcal{O}}(op_j, t - \sigma(op_j)) \preceq \vec{R}_{\mathcal{M}}$$

The local compaction problem bears a lot of similarity with job scheduling and is likewise NP-complete.

3.2 List Scheduling

List scheduling evolved from research in job scheduling theory [Coffman76] and is one of the methods used to perform local compaction. In spite of the theoretical complexity of LCP, extensive experiments have demonstrated that list scheduling is an effective heuristic to obtain near optimal solutions for LCP [Fisher79,Fisher81a,Fisher81b].

List scheduling generates VLIW instructions by performing a topological sort of the DDD. Machine resources like registers and busses are usually assigned on the fly while traversing the DDD. Functional units may however, be assigned by a preliminary pass. The functional unit assignment pass plays the same role as register allocation for conventional compilers, and uses the same kind of algorithms. Preliminary functional unit assignment is done when the target machine is not “symmetric”, i.e. when functional units performing the same operation do not have the same communication capabilities with other machine resources. When the target machine is “symmetric” functional units are assigned on the fly, like any other machine resource. What differentiates the two versions of list scheduling, is the way instructions are generated. In both versions the DDD is traversed in breadth first topological order.

Functional units assigned by a preliminary pass. Instructions are generated one at a time, in sequential order: first the instruction executed at cycle 0, then the one executed at cycle 1, etc. To form the next instruction, list scheduling considers all *data ready* nodes. A node is data ready if all the operations on which it depends (i.e. its predecessor in the DDD) have already been scheduled. The list scheduler fills the current VLIW instruction with as many data ready operations as resource constraints permit. When no more data ready nodes can be squeezed in, the instruction is emitted, and the packing of a new instruction is started.

Functional units assigned on the fly. Several instructions are generated at the same time. When an operation *op* is selected from the data ready list, the earliest cycle (Min_Cycle) in which *op* can be scheduled is computed. Min_Cycle is determined by the data precedence constraints imposed by the already scheduled operations. Based on the resource constraints imposed by the already scheduled operations, we determine Ok_Cycle ($\text{Min\_Cycle} \leq \text{Ok\_Cycle}$), the earliest cycle in which *op* is actually scheduled. If all operations take unit time, or machine resources are infinite, this view of list scheduling is identical to the previous one, that is instructions are created sequentially, and not simultaneously.

When packing operations into VLIW instructions, the code generator is often faced with a choice of several data ready nodes. A commonly used heuristic is the *critical path* heuristic, where operations which lie on the longest path (in terms of execution time) are given priority.

Let $\text{Succ}(op)$ be the set of operations which depend on *op* in the DDD (i.e. the successors of *op* in the DDD), and $\text{Pred}(op)$ be the set of operations on which *op* depends (i.e. the predecessors of *op* in the DDD). Let $\text{Priority}(op)$ be the priority of *op*. Here is the list scheduling algorithm for a “symmetrical” VLIW machine (functional units are assigned on the fly):

```

FUNCTION List_Schedule ( DDD = ( $\mathcal{O}, E$ ) ) RETURN a schedule  $\sigma$  IS
BEGIN
  Data_Ready := { $op \in \mathcal{O} \mid Pred(op) = \phi$ };
  Scheduled :=  $\phi$ ;
  WHILE Data_Ready  $\neq \phi$  LOOP
    let  $op_{hp}$  be the operation with highest Priority in Data_Ready;
    Min_Cycle :=  $\max_{op \in Scheduled} (\sigma(op) + \delta(op, op_{hp}))$ ;
    let Ok_Cycle  $\geq$  Min_Cycle the earliest cycle such that:
       $\forall 0 \leq t \leq d(op_{hp}) \quad \tilde{R}_{\mathcal{O}}(op_{hp}, t) + \sum_{op_s \in Scheduled} \tilde{R}_{\mathcal{O}}(op_s, (Ok\_Cycle - \sigma(op_s)) + t) \preceq \tilde{R}_{\mathcal{M}}$ ;
     $\sigma(op_{hp}) := Ok\_Cycle$ ;
    Scheduled := Scheduled  $\cup \{op_{hp}\}$ ;
    Data_Ready := Data_Ready -  $\{op_{hp}\} \cup \{op_{succ} \in Succ(op_{hp}) \mid Pred(op_{succ}) \subseteq Scheduled\}$ ;
  END LOOP;
  RETURN  $\sigma$ ;
END List_Scheduling;

```

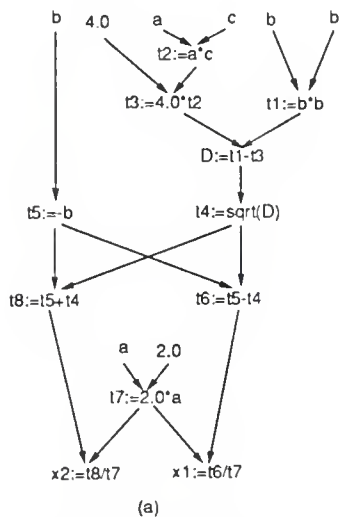
A simple implementation of the algorithm runs in $O(n^2)$ plus the time to compute the *Priority* function for the n operations $\{op_1, \dots, op_n\}$.

Let us now consider the following example. Assume an (unrealistic) machine with one floating point adder, multiplier and square-root/divider units. All these units are pipelined, and take respectively two, two and three cycles to complete an operation. We assume memory and register access time to be null. Consider the following straight line code which computes the roots of a second degree polynomial (we assume $D \geq 0$). Its intermediate representation is given to its right. The DDD is given in figure 4(a). Nodes containing variables have been duplicated for convenience.

	$t1 := b*b$
	$t2 := a*c$
--	$t3 := 4.0*t2$
-- assume $D \geq 0$	$D := t1-t3$
--	$t4 := \text{sqrt}(D)$
	$t5 := -b$
$D := b*b-4.0*a*c$;	$t6 := t5-t4$
$x1 := (-b-\text{sqrt}(D)) / (2.0*a)$;	$t7 := 2.0*a$
$x2 := (-b+\text{sqrt}(D)) / (2.0*a)$;	$x1 := t6/t7$
	$t8 := t5+t4$
	$x2 := t8/t7$

If we use the critical path heuristic for the *Priority* function, the schedule produced by list scheduling is given in figure 4(b).

The previous algorithm schedules operations as early as possible, regardless of the cycle in which an operation result is used. This increases variable lifetime (the time-span between the definition and last use of a variable), and consequently the time-span during which a variable has to be kept in a register. To reduce variable lifetime a bottom-up approach can be used, as suggested by Lam [Lam87] and other researchers [Mueller86]. DDD edges are reversed, and list scheduling is applied to this modified DDD. The schedule



Cycle	Floating +	Floating *	sqrt/division
1	$t5 := -b$	$t2 := a * c$	—
2	—	$t1 := b * b$	—
3	—	$t3 := 4.0 * t2$	—
4	—	$t7 := 2.0 * a$	—
5	$D := t1 - t3$	—	—
6	—	—	—
7	—	—	$t4 := \text{sqrt}(D)$
8	—	—	—
9	—	—	—
10	$t6 := t5 - t4$	—	—
11	$t8 := t5 + t4$	—	—
12	—	—	$x1 := t6 / t7$
13	—	—	$x2 := t8 / t7$

(b)

Figure 4: (a) The data dependence dag. Nodes containing variables are duplicated for convenience. (b) Schedule obtained by applying list scheduling.

obtained will itself have to be reversed to obtain a valid schedule. By using reverse topological ordering, a store into a register is scheduled after all its uses have been scheduled. By placing the store as close as data and resource constraints permit, the lifetime of a variable is shortened. If bottom-up list scheduling is applied to the previous example $t7 := 2.0 * a$ will be scheduled in cycle 10 instead of 4.

4 Global Compaction Techniques

This section focuses on three important global compaction techniques: *trace scheduling* [Fisher84b,Ellis86], *percolation scheduling* [Nicolau85a,Nicolau85b,Aiken88a] and *region scheduling* [Gupta87c,Gupta88].

As previously mentioned, the necessity of such global compaction techniques stemmed from experiments performed in the early 70s. The experiments (section 4.1) showed that very little fine grain parallelism is available within basic blocks, but massive amounts of such parallelism can be uncovered if one disregards the basic block barrier.

Trace scheduling (section 4.2) was one of the first techniques that went beyond basic blocks in its search for fine grain parallelism. This technique employs branching probabilities to make educated guesses on the run time behavior of a program.

Percolation scheduling (section 4.4) is a simplification and generalization of the ideas behind trace scheduling. A program is compacted without the need for run time information by repeatedly applying a series of elementary program transformations.

Region scheduling (section 4.5) redistributes the parallelism available throughout a program, taking into account machine capabilities. Like in percolation scheduling, repetitive application of elementary program transformations is employed. However while percolation scheduling transformations operate on the flow graph of a program, region scheduling employs a new intermediate program representation paradigm: the *program dependence graph* [Ferrante87].

4.1 Amount of Fine Grain Parallelism Available in Serial Programs

Experiments performed in the early 70s [Riseman72,Foster72,Tjaden70] looked at the amount of fine grain parallelism that was available in serial programs. The experimenters examined machine language streams of IBM and CDC machines, broke these streams into basic blocks, and determined through simulation the speedup that one could obtain by using infinite hardware. Memory access time was assumed to be null and renaming was used to avoid unnecessary contention problems. The primary concern of the experiments was the manner in which conditional instructions were treated.

Blocking on conditionals: the first set of experiments explored the parallelism available *within* basic blocks, that is, upon encountering a conditional branch the infinite machine would wait until the condition resolved before scheduling any other operation. The speedup obtained was on the order of 2. This is insufficient to keep a highly parallel architectures busy.

Bypassing conditionals: the second set of experiments assumed the path taken at each branch to be known in advance. Speedups ranging from 8 to over 100, with an average of about 50 were observed. The experiments also considered bypassing a fixed number of conditionals at any given time. The relative speedup increased as $\sqrt{j}$, where j is the number of conditionals bypassed. This relation holds well for $j \leq 32$ [Riseman72]. Foster and Riseman give no theoretical justification for this fact.

Another set of experiments determined the amount of parallelism that could be uncovered if conditional jumps are ignored, but the search for parallelism is limited to a fixed window of instructions. [Riseman72] found that for a window of size 64, only one fourth of the potential speedup could be obtained. This implies that instructions must be extensively rearranged in order to achieve maximum speedup.

The programs tested were symbolic as well as numerical. It is interesting to remark that experimenters found no significant result differences among these two kinds of programs [Riseman72]. This would tend to support the use of VLIWs to speedup non numerical type of code [Ebcioğlu88].

Nicolau and Fisher [Nicolau81,Nicolau84b] did similar experiments, focusing on numerical programs. They also assumed a machine with infinite hardware. An *oracle* was used to compact code for execution on the infinite machine. The role of the oracle was double. It had to predict the outcome of conditional jumps and to resolve all ambiguous memory references. Extensive renaming was used. The execution time of the oracle compacted program on the infinite machine, is just the length of the longest data flow dependence chain in the program. The speedups observed averaged 90.

4.2 Trace Scheduling

4.2.1 The Trace Scheduling Algorithm

One of the first techniques to exploit fine grain parallelism across basic blocks was trace scheduling (TS) [Fisher81a,Fisher81b]. The experiments Nicolau and Fisher performed, were carried out with trace scheduling specifically in mind. The primary goal of TS was to replace the experiments oracle with some sort of compile time predictor. The compile time predictor was obtained by assigning branching probabilities to conditional jumps. By using these branching probabilities, TS can select the most likely execution path of a program (such a path is called a *trace*). Once a trace has been selected, it is compacted as if it were a big basic block. The compaction phase may move operations above or below conditional jumps. To preserve semantic equivalence with the original program, TS has to insert *bookkeeping* code in the points where the trace interfaces with the rest of the program (these connection points are called *joins* and *splits*³). After the most probable trace has been compacted the next most probable trace is selected. The process repeats until all but the most unprobable traces remain. The remaining blocks are compacted using some local compaction algorithm. See Figure 5 for an example. Dashed edges in the DDD are special edges and are similar in nature to control dependence edges (section 2.2). TS has to introduce such edges to prevent illegal code motions (in this case prevent the square-rooting of a negative number, and division by zero). Operations in a single horizontal box can be executed concurrently. Note that in this particular example the bookkeeping nodes are not needed, but there is no way for TS to tell, without extensive analysis. We now describe the three phases of TS in more detail [Fisher84a,Fisher84b,Ellis86].

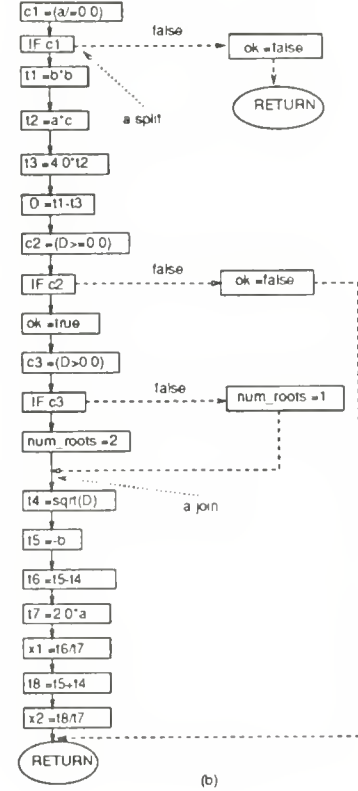
Trace selection. In order to select traces one must gather probabilities about the branching behavior of a program. TS relies on programmer supplied hints, and loop nesting. The programmer can also use an automatic profiler. By using branching probabilities and loop bounds TS selects a seed operation with highest execution probability. The trace is grown back and forward from the seed. To grow forward, the trace picker looks at *op_{last}*, the operation currently at the end of the trace. If *op_{last}* is not a conditional branch the choice of the next operation is obvious. If *op_{last}* is a conditional branch then we select the operation which is executed by following the branch with highest probability. Once the successor operation has been selected it is added to the end of the trace and the process is repeated. A trace is grown backward in an analogous

³A split is a conditional jump out of the trace, a join is a conditional jump into the trace.

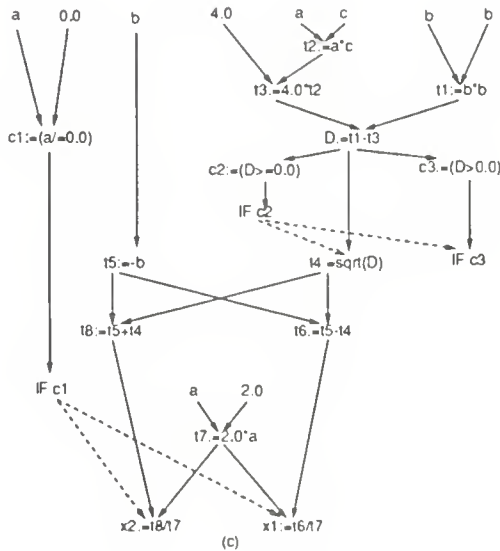
PROCEDURE Roots (a,b,c: IN Float;
x1,x2: OUT Float;
Num_Roots: OUT Integer;
Ok: OUT Boolean) IS

D: Float;
BEGIN
IF (a=0) THEN
Ok:=False; RETURN;
END IF;
D := b * b - 4.0 * a * c;
IF (D < 0.0) THEN
Ok := False;
ELSE
Ok := True;
IF (D = 0.0) THEN
Num_Roots := 1;
ELSE
Num_Roots := 2;
END IF;
x1 := (-b-sqrt(D))/(2.0 * a);
x2 := (-b+sqrt(D))/(2.0 * a);
END IF;
END Roots;

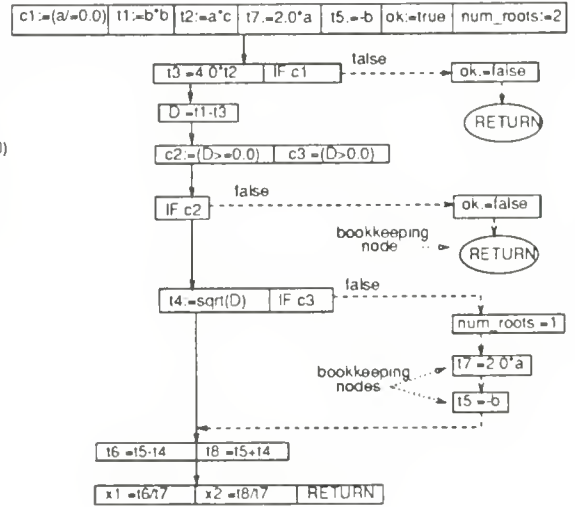
(a)



(b)



(c)



(d)

Figure 5: (a) Initial program. (b) Program flow graph. The trace is composed of the operations connected by solid lines. (c) The data dependence dag of the trace. (d) The program flow graph after compaction and bookkeeping.

fashion. A trace stops growing in one of the two directions when the operation chosen has already been selected by a previously formed trace. Also a trace is not allowed to cross loop boundaries (this restriction was imposed to make memory reference disambiguation simpler). To increase the parallelism of loop body traces, the compiler unrolls inner loops.

Trace Compaction. After a trace has been selected it is handed to the code generator, which compacts it using list scheduling. Because traces span over several basic blocks, special edges (control dependence-like edges) have to be added to the trace DDD to prevent illegal code motions (note that the CDG hadn't been discovered at the time research on trace scheduling was performed). Let cj and op be respectively a conditional jump and an operation. If both cj and op lie on the trace and op clobbers a variable which is live on the off-trace branch of cj , a special edge is inserted in the DDD from cj to op .

After the trace has been compacted and VLIW instructions generated, the code generator deletes the trace from the program flow graph and replaces it with position information. Position information gives the register or memory locations where variables live at the beginning or end of the trace are stored. This information is needed to interface traces.

Bookkeeping. After compacting a trace, the bookkeeping phase has to ensure that semantic equivalence with the initial program is preserved. If an operation op originally below a split is moved above it by the code generator, op has to be copied in the split off-trace branch. The case of a join is similar: a join is placed as early as possible in the compacted code, such that no operation originally above it, is now below it. All operations that have moved above joins have to be copied.

To copy a jump into a split one has to consider the off-trace and the on-trace edges of the conditional jump. When the conditional jump is copied the off-trace edge of the copy is left unchanged while the on-trace edge points to the next operation in the split. Things get a little more complicated when jumps have to be copied up into joins. Consider the operations on the trace that are below the join and above the jump, but that are not copied into the join because they are scheduled below it. These operations have to be copied onto the off-trace edge of the jump's new copy. Note that this rule is *not* applied recursively: if the operations contain conditional branches, they are just copied like any other operation.

4.2.2 VLIW Machines Using Trace Scheduling

The Bulldog (Fortran) compiler designed at Yale implemented TS [Ellis86,Fisher84b]. Its target machine was the ELI (Enormously Longword Instructions) [Fisher83]. This machine was never built and remained a paper design. However some of the people working on the VLIW project at Yale co-founded a company (Multiflow Computers, Inc.) to produce a commercial processor (the TRACE(*TM*) series [Colwell88]) using VLIW and trace scheduling technology. Both the ELI and the TRACE have been designed for scientific computing. We first describe ELI and then its industrial evolution.

ELI's machine resources are divided among 8 identical clusters which are connected by long slow busses (figure 6(a) shows the cluster interconnections). Cluster resources are connected via a partial crossbar; they comprise (see figure 6(b)):

- an integer ALU, a pipelined floating adder and a pipelined floating multiplier. The inputs of each func-

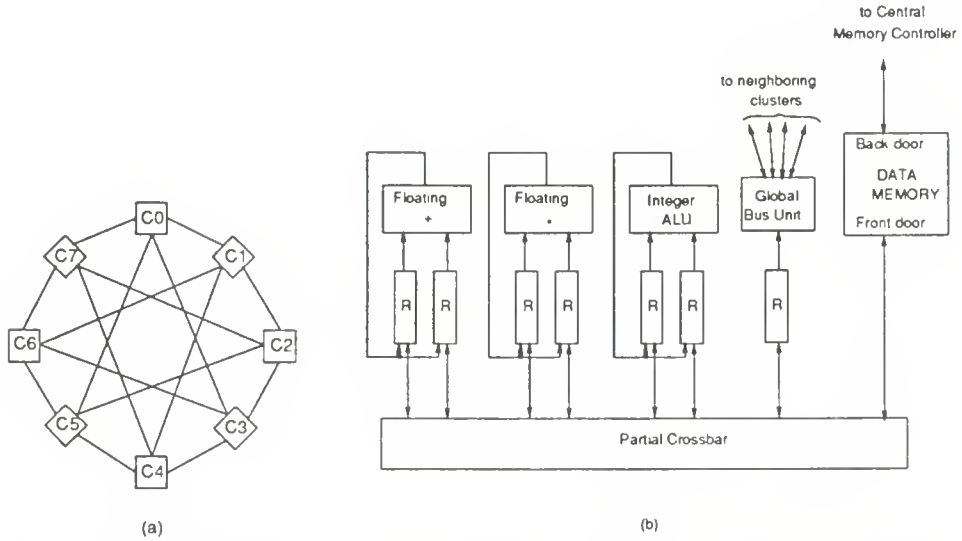


Figure 6: (a) Cluster interconnections of the ELI. (b) The structure of a cluster.

tional unit are connected to two register files of 16 registers each. Division is not directly implemented, a Newton's method approximation is used (an 8 bit inverse is obtained via table look-up);

- a global bus unit that connects the cluster to each of its four neighbors;
- one of 8 dual ported data memory banks. To match processor and memory speeds ELI provides one dual ported data memory bank per cluster. The first port (*front door*) provides direct access to memory while the second port (*back door*) is connected to a central memory controller (located in some of the clusters) which resolves (at execution time) memory references whose bank could not be determined at compile time. A memory bank front door is connected to the cluster partial crossbar. The front door contains a limited integer ALU for address arithmetic, as well as two sets of index registers (not shown in figure 6).

Loop unrolling is one of the techniques the Bulldog compiler uses for memory bank disambiguation. A preloop fixes at run time the initial memory bank of a series of vector references. The remaining iterations are executed in a second loop where unrolling ensures that vector references are bank disambiguated.

The ELI has an instruction word of over 500 bits. Each cluster activity is controlled by a piece of this very long instruction.

As demonstrated by the experiments of section 4.1, basic blocks do not contain enough fine grain parallelism. This implies that a high fraction of program operations are tests. Packing a single conditional jump per VLIW instruction will therefore prevent significant speedups, as a computation running time is bounded by the number of conditionals that have to be executed at run time. Part of ELI very long instruction word is therefore devoted to conditional jumps. Up to k ($k \in \{3, 4, 5\}$) conditional branches can be packed per

VLIW instruction. **Semantically** the conditional part of an instruction executes like a lisp `cond` statement. The k conditional tests are evaluated one after the other. The first test which is true causes a jump to its corresponding branching address. If none of the tests is true control falls through to the next instruction. Of course the **hardware implementation** evaluates all the k conditions in parallel and uses a priority encoder to select the first one that is true. Note that this branching mechanism is closely related to the needs of TS.

The results obtained for the Bulldog-ELI symbiosis are encouraging. An ideal ELI with an infinite number of clusters, where operations take a cycle and intra-cluster communication delays are nil can achieve speedups ranging from 3 (for sequential applications) to 50 (averaging 17)⁴. Most of the applications tested were scientific routines. The results obtained for the (realistic) ELI previously described are speedups up to 8 (which is the maximum speedup since there are 8 clusters), with an average of 5.

While the ELI was originally designed to be a back end processor, the TRACE(TM) series built at Multiflow [Colwell88] is a fully fledged multiuser machine supporting virtual memory and running UNIX. The machine has three hardware configurations, capable of executing 7, 14 or 28 instructions simultaneously. The 14 instructions machine yields 4 to 10 times the performance of a DEC 8700 at twice the cost, and from 1/2 to 1/5 the performance of a Cray-XMP that costs over 20 times as much (these benchmarks are based on Linpack and Livermore Loops). The goals in the design of the TRACE series were to use high volume low cost electronics, DRAMs for main memory, support IEEE 64-bit standard floating point computations, and perform well in a multiuser environment.

The computing power of the machine is provided through four integer and floating point boards (respectively called I and F boards, see figure 7 for details). An I-F board pair corresponds to an ELI cluster. A machine is sold with one, two or four I-F boards, corresponding to a 256, 512 or 1024 bit instruction word. The 1024-bit configuration initiates 28 operations and 4 ELI-like conditional tests (one per I board) for each instruction. The machine peak performance is 60 MFLOPs.

In order to support virtual memory, every machine operation carries along its destination (register or memory) so that when an interrupt occurs the system can complete the current operations and save the computed results, before responding to the interrupt.

In the fully configured machine four memory references (one per I board) can be started every 65 ns. As no data cache is present, memory bandwidth is provided through eight memory controllers each one controlling up to eight independent banks. Memory addresses are fully interleaved. Figure 8 shows the top level architecture of a fully configured TRACE.

The TRACE has a physically distributed instruction cache. There is 1 Mbyte of cache in a fully configured machine. Cache, as well as TLBs, are process tagged, so that no flushing is necessary at context switches. Instruction caches output a fixed length 1024-bit instruction per clock cycle (in a fully configured machine). The bits output by the caches are hardwired to the functional units that they control. Instructions which do not involve floating point computations contain a number of `no-ops`. To minimize the size of the object code stored in primary or secondary storage a variable length representation of the fixed length machine instructions is used.

TRACE runs 4.3 BSD UNIX. UNIX C code has been compiled using a trace scheduling compiler, and has produced good VLIW code, something the Multiflow team did not expect. The size of the UNIX code

⁴The speedup ratios are all measured against the 1 cluster ELI configuration.

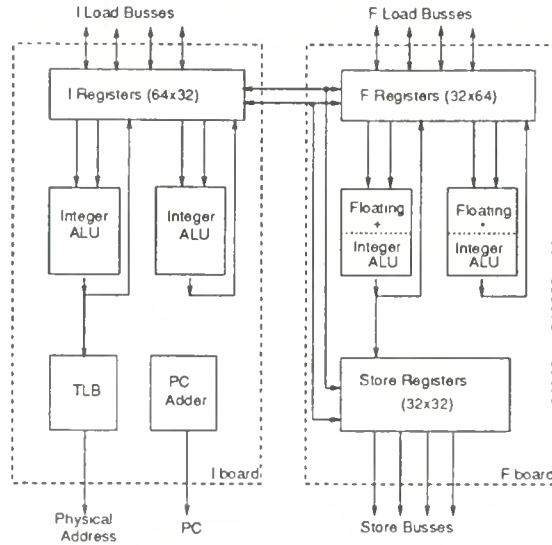


Figure 7: I board and F board block diagrams. Floating point units are pipelined. Physical addresses are generated on I boards, while data to be stored in memory comes from F boards (Store Registers). The 32-bit PC Adder generates branch addresses. I boards can perform an operation every 65 ns, F boards one every 130 ns.

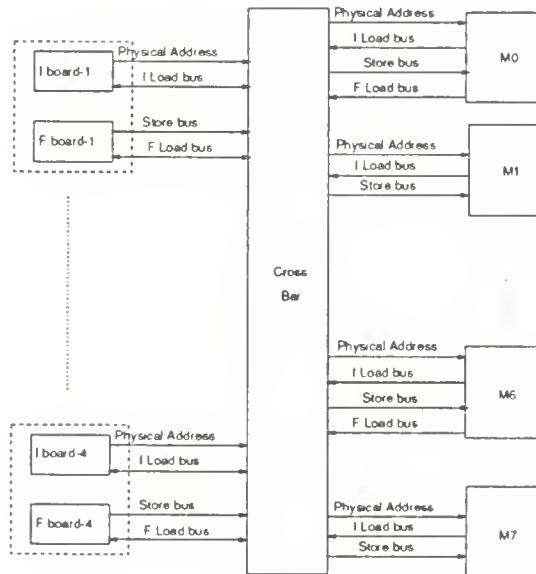


Figure 8: Top level architecture of a fully configured TRACE. The ILoad, FLoad and Store busses are 32 bits wide. Each bus is independently scheduled by the compiler. There is a full crossbar between address generators (I boards) and memory controllers (M0 through M7).

on the TRACE is a factor of 3 bigger than on a VAX (this growth in size is due to bookkeeping).

4.2.3 Problems of Trace Scheduling

TS has various problems, one of the biggest being code explosion. The patching code generated during bookkeeping can potentially be exponential in the size on the input program [Ellis86]. One might indeed wonder whether TS actually terminates, as the generation of bookkeeping code might constantly increase the size of the code still to trace schedule. [Nicolau84a] proved that this is not the case. In practice bookkeeping does indeed increase code size by a substantial amount. If we add in the amount of extra code generated by loop unrolling, code size increase might become problematic. In the experiments that Ellis performed code size increased by a factor of 6 [Ellis86]. The following program fragment illustrates code explosion:

```

FOR i IN 1..n LOOP
  IF abs(a(i)) > ε THEN
    c(i) := b(i)/a(i);
  ELSE
    c(i) := c_max;
    a(i+1) := 2 * a(i+1);
  END IF;
END LOOP;

```

Assuming a 64 banks VLIW machine, the loop body might be unrolled as much as 64 times. The flow graph that we would obtain would then look like figure 9(a). Assuming that the most probable trace is IF c1; o1; IF c2; o2; ... ; IF cn; on, figure 9(b) shows how the code generator might possibly reorder trace operations. Note that although this reordering would normally be illegal for the above loop, provisions for this kind of code motions are made in the TRACE. Figure 9(c) gives the loop after bookkeeping.

Let n be the number of conditional branches in the initial loop body, and let $f(n)$ be the code size of the loop body after the *entire* loop has been trace scheduled. We have:

$$f(n) = 2n + (f(n-1) + 1) + \dots + (f(1) + 1) + f(0)$$

where $f(0) = 1$. By differencing and substitution we obtain: $f(n) = 3 \cdot 2^n - 3$ ($n \geq 1$). Ellis [Ellis86] has showed that $O(n!)$ increase in code size is possible.

Inferring a program run time behavior is another problem of TS. While branching probabilities are skewed for numerical code, 50–50 branching behavior is more likely to be observed for symbolic programs. This restricts the applicability of TS to scientific code (although the Multiflow team employed TS for system code). It is possible however, to apply non trivial global code motions without any knowledge of a program run time behavior [Nicolau85a,Gupta87c,Gupta88]. This leads to a more structured global code motion strategy which avoids some of the useless bookkeeping code introduced by TS (like in the example of figure 5).

The unsatisfactory treatment of loops is another major problem of TS. As traces are limited to loop inner bodies, loop unrolling has to be employed to obtain sufficiently long traces. Since performance usually improves with the amount of unrolling, speeding loops conflicts with the need of preventing code explosion.

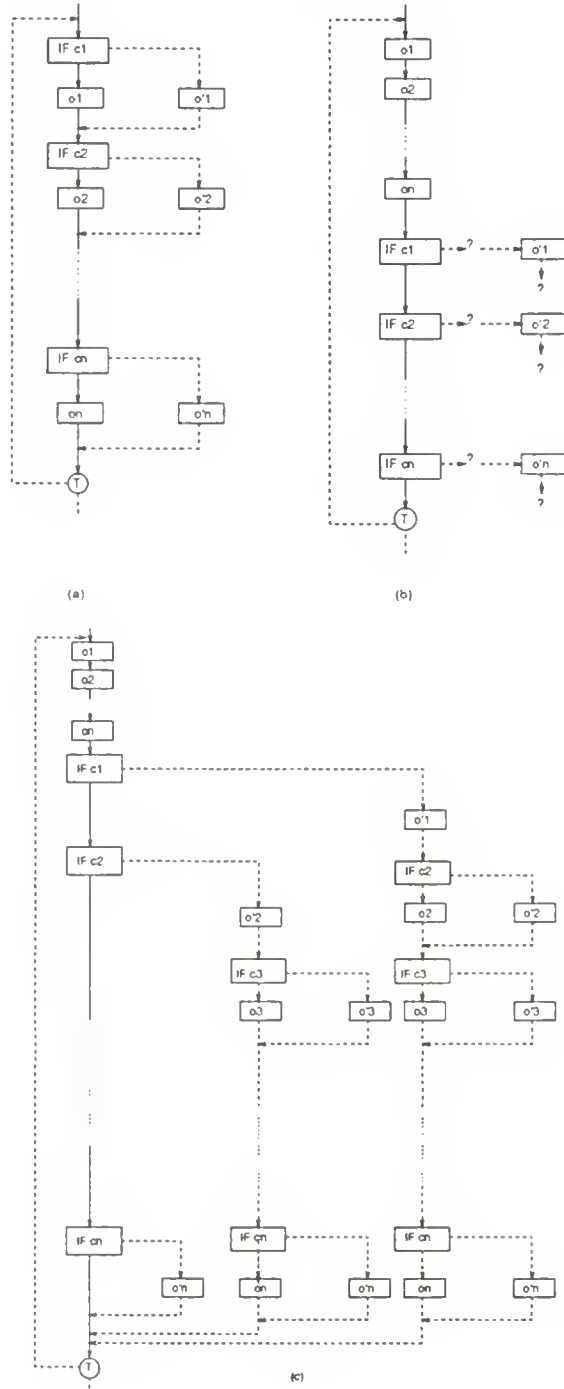


Figure 9: (a) Example of a flow graph which produces code explosion. (b) Possible reordering of trace operations made by the code generator. (c) The loop after bookkeeping.

4.3 Improvements of Trace Scheduling

In this section we survey various improvements of TS.

- Tree compaction [Lah83] avoids code explosion, as well as TS bookkeeping complexity.
- SRDAG compaction [Linn83], is a generalization of TS. It is a first attempt to use more global information in making compaction decisions.

Another improvement of TS is due to Su. Su [Su84] gives a method that provides an alternative treatment of loops. His URCR (UnRoll Compact Reroll) method was developed in the context of microcode compaction. The idea behind URCR has been subsequently adapted to serial programs [Su86,Su87], and is presented in section 5.

4.3.1 Tree Compaction

Tree compaction works like TS except that the schedule range is limited. A program is partitioned into *top* and *bottom* subtrees to which TS is separately applied. Operations are not allowed to be moved passed the root of the subtree they are in. This limits the number of operation copies that have to be made during bookkeeping.

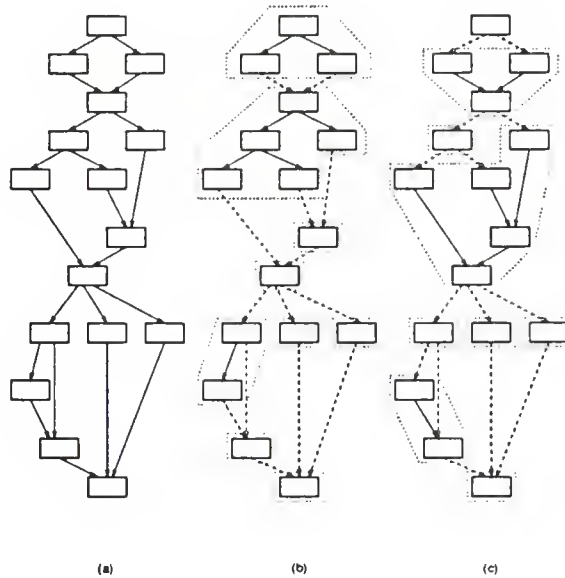


Figure 10: (a) A program flow graph. (b) The top trees of the flow graph. Top trees are surrounded by dotted lines. (c) The bottom trees of the flow graph.

The top trees of a flow graph are determined as follows (bottom trees are determined by using the same algorithm on the reversed flow graph).

1. Start at the entry block of the flow graph. Mark it a top tree root.
2. Once the root r of a top tree has been determined perform a depth first search starting at r .
3. Stop growing the tree in the current direction as soon as you hit a vertex v with $out-degree(v) > 2$ or with $in-degree(v) > 1$; enqueue such a vertex v in a queue Q (if v not already in Q).
4. Once the current top tree has been fully grown dequeue the first element of Q , mark the vertex as a top tree root and repeat steps 2 and 3.

See figure 10 for an example. After the blocks of the flow graph have been partitioned into top trees, TS is applied individually to each top tree. Let $\mathcal{P}_{top}$ be the program obtained after all the top trees have been compacted. The final code is obtained by building $\mathcal{P}_{top}$'s flow graph, finding its bottom trees and applying TS individually to each bottom tree. Note that like in TS traces never cross loop boundaries as top and bottom trees never contain back edges.

The worst case space requirements of the algorithm are quadratic in the number of conditional branches present in the program [Lah83]. Experiments [Su85] performed in the realm of microcode programs showed that tree compaction has indeed better space requirements than TS, and that the space saved does not entail a significant loss in execution speed. In addition tree compaction executes faster than TS since the complexity of list scheduling and bookkeeping are lower, as traces are shorter. These results do not carry over directly to serial programs, but give nonetheless some indications.

4.3.2 Singly Rooted Dag Compaction

The singly rooted dag (SRDAG) method is based on the observation that traces should be reselected after the first block of the current trace has been compacted. The reason is that once a conditional branch has been moved up and scheduled in the first block, the most likely trace in the program might have changed. While TS limits its global view of a program to simple flow graph paths, the SRDAG algorithm considers whole singly rooted dags. The advantage over TS is that there is a more global view. For example an operation might be present at the top of several basic blocks in the SRDAG. The SRDAG algorithm prefers to move such duplicated operations up, whereas TS is not aware of this possible improvement. The SRDAG algorithm is iterative. An iteration comprises 3 steps:

1. **SRDAG selection.** Use the program probability function to determine which uncompact basic block B_{root} (with no uncompact predecessors) has highest execution probability. The SRDAG is the maximal singly rooted dag, rooted at B_{root} , comprising only uncompact blocks (note that SRDAGs never cross loop boundaries).
2. **Compaction.** Create an empty basic block $B_{compacted}$ above B_{root} . Use list scheduling to move operations from an arbitrary block in the SRDAG to $B_{compacted}$. List scheduling is modified so that multiple data ready copies of an operation are represented by a single instance of the operation (a list of pointers is maintained to locate the instances of the operation). The priority of the representing operation is the sum of the individual priorities. Being able to schedule this set of identical operations all at once is the global-view advantage of the SRDAG algorithm over TS. The compaction step ends as soon as a conditional branch is scheduled in $B_{compacted}$.

3. **Bookkeeping.** Like for TS the compaction phase necessitates the introduction of bookkeeping code to preserve semantic equivalence with the original program. The bookkeeping algorithm is similar to that of TS, but is complicated by the fact that in the SRDAG algorithm we are dealing with dags instead of paths. See figure 11 for an example.

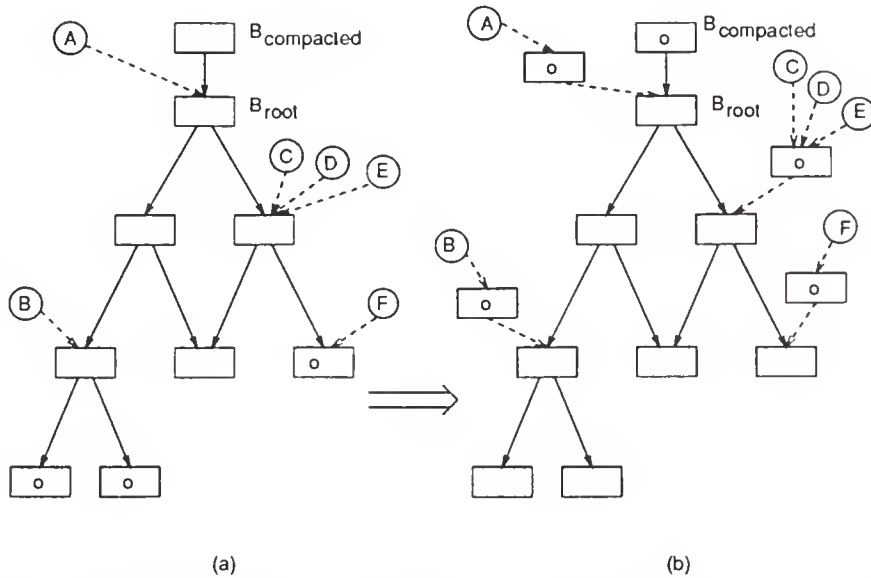


Figure 11: (a) The original SRDAG. o is the operation to be scheduled in $B_{compacted}$. (b) The flow graph after bookkeeping.

The algorithm iterates until the whole program has been compacted (but no proof that the algorithm terminates is provided). Linn conjectures that code compacted with the SRDAG method will never run more than a factor 2 faster than trace scheduled code.

As it is stated, the algorithm allows only a single conditional jump per (very long) instruction. This restriction can be removed by doing the following. As soon as a conditional branch is moved in $B_{compacted}$ a special mode is entered where only conditional branches are allowed to move in $B_{compacted}$. The compaction step ends when no conditional jumps can be moved in $B_{compacted}$.

The major criticism that can be formulated against the SRDAG algorithm is having to consider a singly rooted dag for every block that is compacted. This entails a serious execution overhead, which was confirmed by the experiments performed by Su [Su85] in the context of microprogramming.

4.4 Percolation Scheduling

Percolation scheduling (PS) [Nicolau85a,Nicolau85b,Aiken88a] is a global compaction technique developed by Nicolau as a result of his experience with trace scheduling [Nicolau84a]. PS is a simplification and generalization of the ideas behind TS, transformations have been formalized and proved correct [Nicolau85a].

PS is a hierarchy of semantic-preserving transformations defined in terms of an abstract VLIW machine model with infinite resources (section 4.4.1). *Core transformations* (section 4.4.2) are the lowest level in the PS hierarchy. Higher levels in the hierarchy guide the uniform application of core transformations to “percolate” operations towards the top of the program (hence the name of percolation scheduling) (section 4.4.2). An implementation of the VLIW machine model of PS is presented in section 4.4.3.

4.4.1 The Abstract Machine Model

The abstract VLIW machine $\mathcal{M}$ put forward by Nicolau has infinite resources. An instruction I is of the form:

$$L_I : [Op_I, Cj_I, Next_I]$$

where L_I and $Next_I$ are instruction labels, Op_I is a set of RISC-like operations, and Cj_I is a set of side-effectless conditional jumps⁵ constrained to form a decision tree. $\mathcal{M}$ can execute one instruction every cycle. Instruction execution proceeds as follows:

1. $Op_I \neq \phi$: the operations in Op_I are evaluated in parallel by first reading all the needed variables *and then* performing all the writes. This is consistent with the way registers handle simultaneous reads and writes. Op_I is not allowed to contain two operations which write to the same variable.
2. $Cj_I = \phi$: the label of the next instruction is given by $Next_I$.
3. $Cj_I \neq \phi$: the label of the next instruction is determined by the path in the decision tree which is true.

An instruction I can be portrayed by a binary tree T_I [Ebcioglu88]. T_I 's root is labeled L_I , and each of its leaves is marked with the label of some instruction to which I could potentially branch. Each internal node is labeled with a conditional jump. The interior nodes are organized according to the binary tree-structure imposed by Cj_I . If Op_I is not empty T_I 's root has degree one and its outgoing edge is labeled with the operations in Op_I . The following instruction for instance, can be represented by the tree in figure 12.

$$L_1 : [\{ c5 := (i < n); b := x(i); i := i + 1 \}, \\ \{ \text{IF } c1 \text{ THEN} \\ \quad \text{IF (NOT } c2) \text{ THEN } L_2 \\ \quad \text{ELSE } L_3 \\ \text{ELSE } L_7 \}, -]$$

A program of $\mathcal{M}$ is represented by a flow graph where each vertex contains a single $\mathcal{M}$ instruction. Such a representation is termed a *parallel program graph*. Note that there may be more than two edges out of a parallel program graph vertex as a result of the decision tree structure of the instruction contained in it.

⁵ Conditional jumps are only allowed to test boolean variables, that is tests of the form $\text{if } x = 0 \dots$ are illegal in the PS formalism, and have to be converted to $c := (x = 0)$, $\text{if } c \dots$. This is a standard approach in the VLIW world.

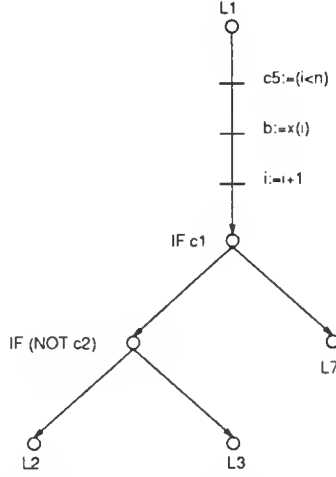


Figure 12: Tree representation of the previous instruction.

The main difference between the model adopted by Nicolau and the model used in the ELI, is the way conditionals are handled. In the ELI only a *vine* of conditionals is allowed, whereas the PS model allows for an arbitrary decision tree (see figure 13).

4.4.2 The Core and Scheduling Transformations

PS is composed of several program transformation layers. The goal of the high level layers is to rearrange a program so that core transformations (the lowest PS layer) are readily applicable. Example of high level transformations are variable renaming, loop interchange, etc. This section will only present the bottommost 2 layers: the core and scheduling transformation layers.

In the remainder of this section we assume acyclic program graphs, PS can easily be extended to handle cyclic code by preventing core transformations to cross back edges. Loops can be compacted by unrolling and applying PS. The section on software pipelining will introduce better techniques (some based on PS) to handle loops.

Core Transformations. Core transformations operate on parallel program graphs, they are extremely low level. There are four core transformations: Delete, Move-op, Move-cj and Unification. In what follows the term “node” refers to a node in the input parallel program graph.

1. **Delete Transformation.** The delete transformation removes unreachable or empty nodes from the parallel program graphs (a node n is empty if $Op_n = Cj_n = \phi$). These kind of nodes are created as a result of other transformations. See figure 14(a).
2. **Move-op Transformation.** The move-op transformation moves an operation op from a node n_2 to a predecessor node n_1 . This transformation can be performed if no true or output data dependences

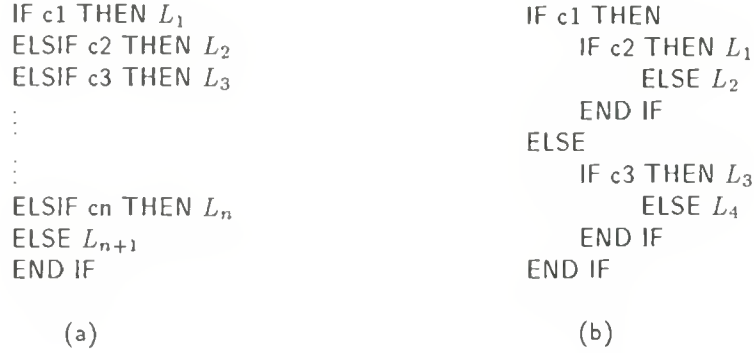


Figure 13: (a) Vine of tests as handled by ELI's multiway jump mechanism. (b) Decision tree that cannot be handled by the ELI.

exist between the operations in n_1 and op . Also op cannot be moved in n_1 if op writes a variable live in any of the successors of n_1 (unless renaming is used). Note that due to the semantics of the abstract machine, anti-dependences do not prevent move-op to take place. Care must be taken to ensure that those paths which go through n_2 , but not through n_1 , are unaffected. To enforce this constraint, n_2 's incoming edges (except for (n_1, n_2)) are set to point to a copy of n_2 . See figure 14(b).

3. **Move-cj Transformation.** The move-cj transformations moves a conditional jump cj from a node n_2 to a predecessor node n_1 provided no flow data dependences exist between the operations in n_1 and cj (remember that cj has no side-effects). A compensation node is created, to ensure that the paths passing through n_2 but not through n_1 are unaffected by the transformation. See figure 14(c) for a detailed description of the transformation.

4. **Unification Transformation.** The unification transformation is similar to code hoisting. Assume the successors $n_1, \dots, n_k$ of a node m all contain the same operation op . op can be simultaneously moved from $n_1, \dots, n_k$ to m if there is no true or output data dependence between op and any of m 's operations. The unification transformation can be performed even when some of the successors of m (call them $K_1, \dots, K_t$) do not contain op in their operation set, provided op can be inserted in $K_1, \dots, K_t$ without modifying the program semantics, that is the variable written by op is not live in the nodes reachable from $K_1, \dots, K_t$. As usual paths passing through any of the n_i s but not through m must be unaffected, and appropriate node copies have to be inserted. See figure 14(d). Note that the unification transformation degenerates to the move-op transformation for $k = 1$.

Scheduling Transformations. To apply core transformations we need some sort of high level guidance. Scheduling transformations are responsible for compacting a program. We will introduce the two most useful scheduling transformations: *migrate* and *greedy-compact*⁶.

⁶The *greedy-compact* transformation is a modification by Ebcioğlu [Ebcioğlu88,Ebcioğlu87] of the original *compact-path*

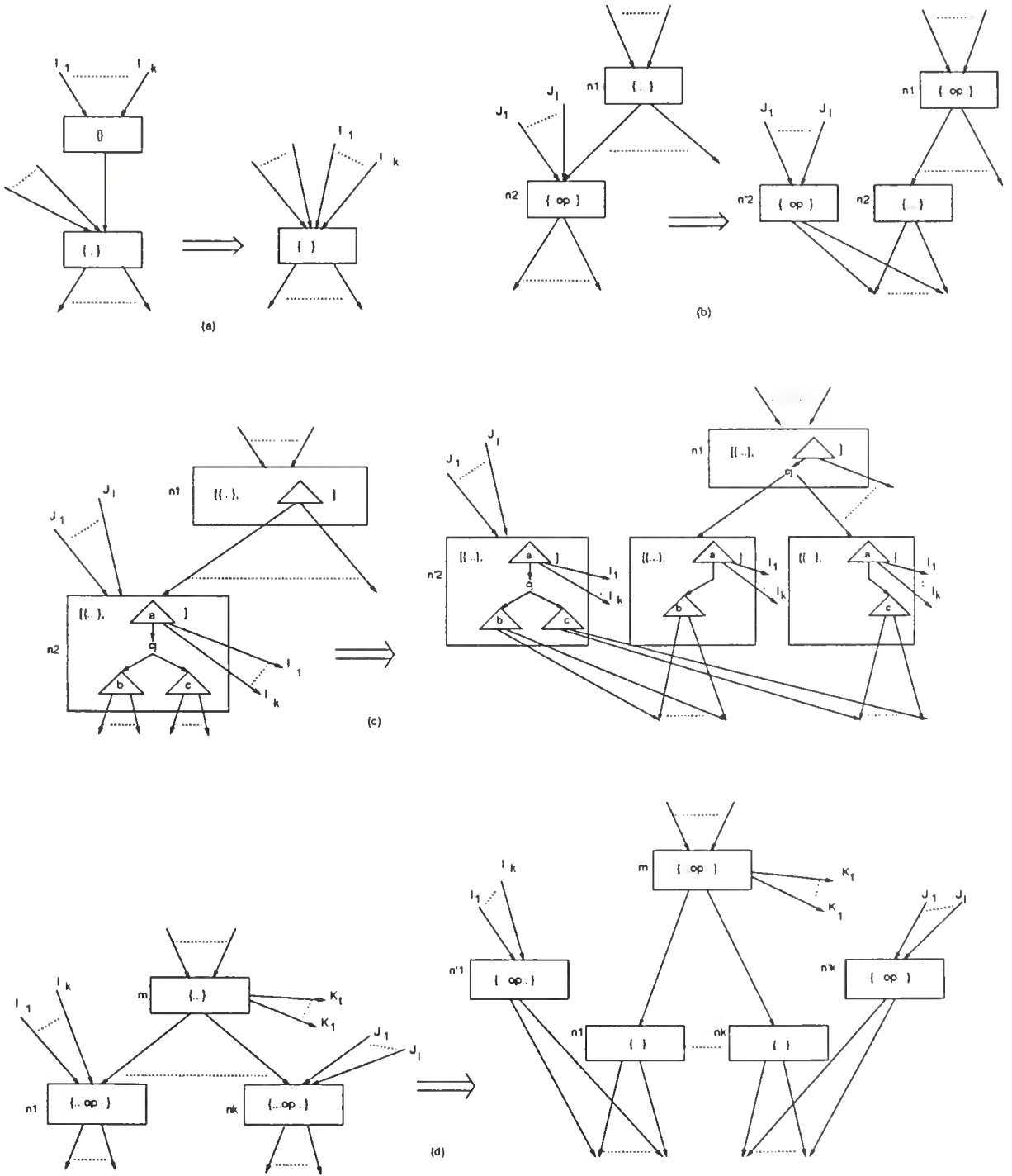


Figure 14: Only edges which are actively involved in transformations are labeled. (a) Delete transformation. (b) Move-op transformation. (c) Move-cj transformation. (d) Unification transformation.

- **Migrate Transformation.** *Migrate*(*op*, *n*₂, *n*₁) moves an operation *op* or a conditional jump *cj* as far up in the parallel program graph as data dependences permit. Before moving an operation *op*, from node *n*₂ to node *n*₁ we move up all the copies of *op* that are in nodes reachable from *n*₁ (deepest first). Unifications are performed whenever possible, not only to reduce the risks of code explosion, but also to allow self dependent operations to move up simultaneously. If copies of *op* can be moved to nodes adjacent to *n*₂, unification is invoked to move *op* from *n*₂ to *n*₁, otherwise move-op is used. Renaming transformations are used to enhance the applicability of core transformations.
- **Greedy-compact Transformation.** Greedy-compact is basically a greedy application of the migrate transformation. The parallel program graph is explored, level by level, until all operations have been pushed up as much as possible, with the migrate transformation.

As an example the code given in figure 5(b) is compacted with greedy-compact.

```

L0: [{c1:=(a≠0.0); t1:=b*b; t2:=a*c; t7:=2.0*a; t5:=-b; Ok:=True; Num_Roots:=1},φ,L1]
L1: [{t3:=4.0*t2},{IF c1 THEN L2 ELSE L3 },—]
L2: [{Ok:=False},φ,Exit]
L3: [{D:=t1-t3},φ,L4]
L4: [{c2:=(D<0.0); c3:=(D>0)},φ,L5]
L5: [φ,{IF c2 THEN L6
      ELSE
      IF c3 THEN L7 ELSE L8},—]
L6: [{Ok:=False},φ,Exit]
L7: [{Num_Roots:=2},φ,L8]
L8: [{t4:=sqrt(D)},φ,L9]
L9: [{t6:=t5-t4; t8:=t5+t4},φ,L10]
L10: [{x1:=t6/t7; x2:=t8/t7},φ,Exit]
Exit:

```

Note that as a result of unification operations *t7:=2.0*a* and *t5:=-b* do not have to be duplicated in *L*₇.

Percolation scheduling will always produce code which is as good as the one generated by TS, but with the advantage that less compensation code is generated. PS does not rely on any kind of branching probabilities, programs are compacted by executing operations as early as possible.

4.4.3 ROPE

To take full advantage of PS, Karplus and Nicolau [Karplus85,Karplus86] have designed a machine (ROPE) implementing the decision tree type of branching introduced by the parallel program graph model.

As the machine has remained a paper design the exact structure of the data path is of little importance and is similar to that of the archetype VLIW machine presented in the introduction (figure 15). No interrupt handling for fast context switching is provided, ROPE uses a cheap slow processor to handle instruction traps, page faults and I/Os. The CPU might have to be frozen if the slow processor has not returned the instructions or data needed in time.

transformation of Nicolau. While compact-path uses branching probabilities, and concentrates on compacting a single path, greedy-compact has the advantages of compacting the whole program, and needs no branching probabilities.

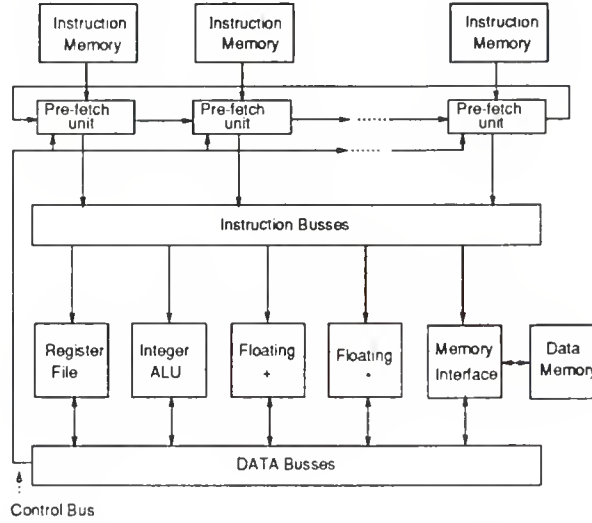


Figure 15: ROPE block diagram. There are 2^k instruction memory banks and pre-fetch units (k is a machine parameter).

What makes ROPE an original design is the way multiple conditional jumps are handled. ROPE was designed to use slow inexpensive memory (dynamic RAM) to support the multiway branching capability of PS, instruction caches are not used. The program memory has to provide an instruction every cycle. Since the cycle time of dynamic RAMs might be as much as 10 times slower than that of the CPU, instruction memory banks are interleaved. This standard approach works well for straight-line code, but imposes untolerable delays when a jump occurs. To avoid jump penalties, each of the 2^k memory banks is connected to an intelligent pre-fetch unit. Pre-fetch units are connected in a ring (ROPE stands for Ring Of Pre-fetch Elements, see figure 15).

Pre-fetch units can be *active*, *ready*, *busy* or *idle*. A pre-fetch unit is *active* when the current instruction is being output, *ready* when memory fetches have completed (and the pre-fetch unit is waiting its turn to output the instruction), *busy* when memory reads are in progress, and is otherwise *idle*.

For straight line code a pre-fetch signal is passed in cyclic fashion from pre-fetch unit i to pre-fetch unit $(i + 1) \bmod 2^k$. To avoid delays this signal has to be issued well before instructions are executed. Let d be the ratio (instruction memory/CPU) cycle time. Once the pre-fetch signal is sent to unit 0 and propagated from one pre-fetch unit to the next one is able to obtain one instruction per cycle after a delay of $d - 1$ cycles (provided $d \leq 2^k$).

In order to process arbitrary jumps efficiently, pre-fetch units are informed ahead of time that they may be requested to output a specific instruction. For instance consider an instruction I containing the following multiway branch:

```

IF c1 THEN
  IF c2 THEN Addr1
    ELSE Addr2
  END IF
ELSE
  IF c3 THEN Addr3
    ELSE Addr4
  END IF
END IF

```

where $Addr_1, \dots, Addr_4$ are the four target addresses; d cycles before instruction I is scheduled, the compiler schedules four special pre-fetches (which we will call jump-pre-fetches) to pre-fetch units $(Addr_1 \bmod 2^k)$, $(Addr_2 \bmod 2^k)$, $(Addr_3 \bmod 2^k)$ and $(Addr_4 \bmod 2^k)$. The compiler has to ensure that these four units are different; code duplication may be necessary. Each of the four jump-pre-fetched units receives not only an address, but also a condition mask (jump-pre-fetched unit $(Addr_3 \bmod 2^k)$, for instance receives $((\text{NOT } c1) \text{ AND } (c3))$ as a condition mask). As soon as instruction I executes the current values of the condition bits are output on the control bus (see figure 15). The jump-pre-fetched unit whose condition mask matches the value on the control bus will output the instruction it has fetched. If the target unit is still busy it will freeze the CPU. Once the current instruction has been generated, control is passed to the pre-fetch unit following the target (assuming no jumps in the current instruction). Thus a jump-pre-fetched unit has to signal to the units to its right that they may be requested to output an instruction in the coming cycles. This is achieved by generating a normal pre-fetch pulse (transmitted from one unit to the next) one cycle after the jump-pre-fetch is received.

Let us compute the number of pre-fetch units that are needed to support an m -way jump. On the cycle after the condition codes have been put on the control bus, each of jump-pre-fetched units has to be ready (since we don't know in advance which will be the target). Furthermore each of the $d - 1$ units to the right of the jump-pre-fetched units must have started pre-fetching. We therefore need $m + m * (d - 1) = m \cdot d$ pre-fetch units. Karplus and Nicolau suggest that a machine with 32 or 64 pre-fetch units would be appropriate.

As the compiler should be allowed to schedule several sets of jump-pre-fetches simultaneously, each jump-pre-fetched units is transmitted a label along with the address and control mask. This label is the same for each of the jump-pre-fetched units involved in a same multiway branch. When the condition bits are output on the control bus, the label of the corresponding multiway branch is also transmitted to the pre-fetch units. The unit which has a match for both the label and condition mask is allowed to output the instruction it has fetched. This insures that no conflicts exists between two jump-pre-fetched units which happen to have same condition mask, but relate to different instructions.

4.5 Region Scheduling

While TS and PS operate on program flow graphs, *region scheduling* (RS) [Gupta87c,Gupta88,Gupta87a] manipulates program dependence graphs (PDG) (section 2.2). The computational model used is simpler than TS' or PS': instructions are allowed a single branch operation. RS works by “fairly” redistributing the parallelism available in a program. Let $Op(R_i)$ be the set of operations directly depending on a PDG region node R_i and let $H(R_i)$ be the height of the data dependence dag of the operations in $Op(R_i)$. Define $\pi(R_i)$, the parallelism present in a region node R_i , to be:

$$\pi(R_i) = \frac{\text{number of operations in } R_i}{\text{longest data dependence chain of operations in } R_i} = \frac{|Op(R_i)|}{H(R_i)}$$

$\pi(R_i)$ is the average number of operations that can be executed in parallel in any given cycle for region R_i .

RS works by repeatedly transforming the PDG, redistributing the parallelism available in the program from region nodes containing high levels of parallelism to region nodes containing low levels of parallelism. The maximum number of operations that a machine $\mathcal{M}$ can actually perform per cycle is determined by the architecture, and is denoted $\pi(\mathcal{M})$. The goal of RS is to make $\pi(R_i)$ approach $\pi(\mathcal{M})$ for all regions, so that system resources are used efficiently. Structurally RS is similar to PS: a set of core transformations (called *region transformations*) are repeatedly applied using higher level guidance rules.

4.5.1 Region Transformations

There are four kinds of region transformations: region deletion, loop splitting/unrolling, forward/backward code motion and region copying/collapsing. As in TS and PS, some transformations require the introduction of compensation code to preserve semantic equivalence with the original program. A description of each region transformation follows.

1. **Region Deletion.** If a region R_i has a single incoming edge $e = (R_j, R_i)$, the deletion transformation $T_{del}(R_i)$ can be applied and region node R_i can be deleted (see figure 16(a)). Redundant regions appear as a result of other transformations.
2. **Loop Splitting/Unrolling.** FOR loops containing no conditionals and having known loop bounds can be split or unrolled. $T_{split}(L_i, R_i)$ splits the loop in region L_i into two loops: one with m iterations is attached to region R_i , and the other with $n - m$ iterations (where n is the known number of initial iterations) remains attached to L_i (see figure 16(b)). In order for T_{split} to apply we must initially have:

$$\pi(R_i) < \pi(\mathcal{M}) < \pi(L_i)$$

The amount of splitting, m , is the first integer ($\leq n$) such that the above double inequality no longer holds. Standard loop unrolling ($T_{unroll}(L_i)$) can be used to increase the parallelism within a loop body. The amount of unrolling performed by T_{unroll} is determined by the machine $\pi(\mathcal{M})$, and the parallelism already available in L_i (the amount of unrolling u is the biggest integer such that after unrolling the loop u times $\pi(L_i) \leq \pi(\mathcal{M})$). This is different than what is done in TS where the amount of unrolling is arbitrary.

3. **Forward/Backward Code Motion.** Forward/backward code motion transformations move code from regions with excess parallelism to regions with insufficient parallelism. This type of transformation can only be applied to *structured* nodes. A region node is structured if the code represented by the subgraph rooted at the node is structured. $\mathcal{T}_{move}(R_j, R_i)$ moves code forward or backward from a region node R_j to an adjacent region node R_i (see figure 16(c) and (d)). It can be applied only if:

$$\pi(R_i) < \pi(\mathcal{M}) < \pi(R_j)$$

The set of operations that can be moved is determined by data dependences. Operations are moved from R_j to R_i until the above inequality no longer holds.

4. **Region Copying/Collapsing.** Region copying/collapsing transformations merge regions with insufficient parallelism. The region copy transformation, $\mathcal{T}_{copy}(R_i)$ copies code from a region R_i into each of R_i 's predecessors (figure 16(e)). This transformation eliminates the need for an unconditional jump instruction. The region collapsing transformation, $\mathcal{T}_{merge}(R_i, Test, R_j)$ merges R_i , $Test$, R_j , to form a new operation node op_k (figure 16(f)). After fusion the operations in regions R_i , R_j and $Test$ are treated as a single unit in further transformations. The merge transformation is applied only if $\pi(R_i) + \pi(R_j) \leq \pi(\mathcal{M})$, and if the architecture allows discarding of values (see section 5.2.1), which would allow the operations in R_i and R_j to execute concurrently. The outcome of $Test$ would decide which one of the two sets of operations would be aborted.

4.5.2 Guidance Rules for Applying Region Transformations

The above transformations can be applied repeatedly to increase the parallelism present in a region up to the point where machine resources are exhausted. The repeated application is necessary because a transformation may enable the application of another transformation. The RS algorithm has two phases: a local and a global transformation phase.

Local transformation phase. The local transformation phase considers adjacent region nodes, and needs only a local view of the PDG. It proceeds as follows:

1. Let $S = \{R_i | \pi(R_i) \leq \pi(\mathcal{M}) - \epsilon\}$, where ϵ is some architecture dependent threshold. S is the set of region nodes with insufficient parallelism.
2. For all $R_i \in S$, apply one of the following transformations in the order given below.
 - (a) $\mathcal{T}_{split}(L_i, R_i)$ if R_i has an adjacent loop region L_i with too much parallelism;
 - (b) $\mathcal{T}_{unroll}(R_i)$ if R_i is a loop region node;
 - (c) $\mathcal{T}_{move}(R_j, R_i)$ if R_i has an adjacent region R_j with too much parallelism;
 - (d) $\mathcal{T}_{copy}(R_i)$
 - (e) $\mathcal{T}_{merge}(R_i, Test, R_j)$ if $\pi(R_i) + \pi(R_j) \leq \pi(\mathcal{M})$.

If the transformation applied hasn't transferred enough parallelism R_i remains in S , and is subsequently reconsidered for other transformations.

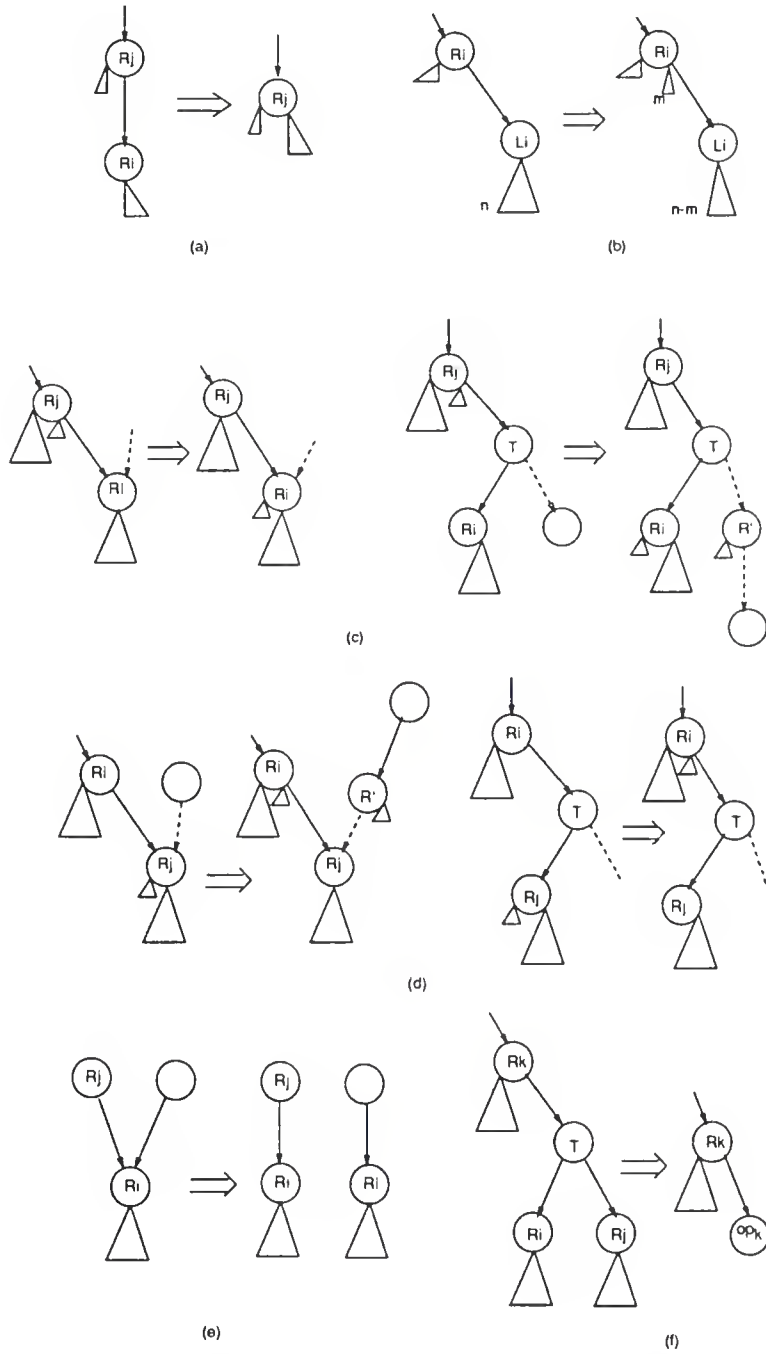


Figure 16: (a) Region deletion. (b) Loop splitting. (c) Forward code motion. (d) Backward code motion. (e) Region copying. (f) Region collapsing.

3. After performing one of the above transformations, T_{del} might have to be invoked to clean up the PDG.
4. Repeat step 2 until $S = \phi$ or no transformations apply.

Global transformation phase. This phase is invoked once local transformations have been applied. Global transformations consider the *unbalanced* paths of the PDG. A path P is said to be unbalanced if its endpoints are two regions nodes, one with insufficient parallelism and the other with excess parallelism. The global transformation phase proceeds as follows:

1. Let $S = \{R_i | \pi(R_i) \leq \pi(\mathcal{M}) - \epsilon\}$.
2. For all $R_i \in S$ find the shortest path $P = R_i, R_{i+1}, \dots, R_k$ such that $\pi(R_k) > \pi(\mathcal{M})$ or $\pi(R_k)$ can be made so by applying T_{split} , T_{unroll} , T_{copy} or T_{merge} locally to R_k . Apply the series of transformations:

$$T_{move}(R_k, R_{k-1}), T_{move}(R_{k-1}, R_{k-2}), \dots T_{move}(R_{i+1}, R_i)$$

R_i remains in S as long as $\pi(R_i) \leq \pi(\mathcal{M}) - \epsilon$.

3. Clean up the PDG by means of T_{del} .
4. Repeat step 2 until $S = \phi$ or no transformations apply.

After global transformations have been applied code size might increase exponentially, but the code explosion danger is not as acute as in TS.

4.5.3 From Region Scheduling to Code Generation

After the PDG has been transformed and the parallelism redistributed with RS, the code generator can schedule the individual regions and generate code for the target architecture. Code generation is carried out one region at a time. The order in which regions are processed is the reverse topological order imposed by the CDG (back edges are ignored). Code for a region node R_i can be generated by using list scheduling on the operations that are directly control dependent on R_i .

Gupta has used RS to compile programs for a reconfigurable LIW (RLIW) architecture [Gupta87b, Gupta87a]. The peculiarity of the RLIW architecture developed by Gupta resides in the ability to reconfigure connections between ALUs in each cycle. ALUs are disposed in an $n \times m$ grid. The ALUs in row i ($0 \leq i < n - 1$) are connected to the ALUs in row $i + 1$ via a unidirectional crossbar. Each column j ($0 \leq j < m$) of ALUs is connected to memory module j (there are as many memory modules as there are columns of ALUs). Data flows uniformly from the first ALU row to the last. Connections between successive rows can be changed every cycle. Memory modules are all connected to an “intermemory module transfer request handler” whose role is to transfer data from one memory module to another when memory disambiguation has failed at compile time.

5 Software Pipelining

All compilation techniques seen so far exploit loop level parallelism by using loop unrolling in conjunction with global compaction. While this simple approach is sometimes adequate, it is in general not the best way to speedup loops. Software pipelining proposes to transform loop bodies so that loop iterations can be started every cycle. The name software pipelining comes from the similarity with hardware pipelining, where operations are initiated on a cycle basis. Figure 17 shows a three stages hardware pipeline and its time diagram. If we consider a three instruction loop such as the one in figure 18(a) as a three stages

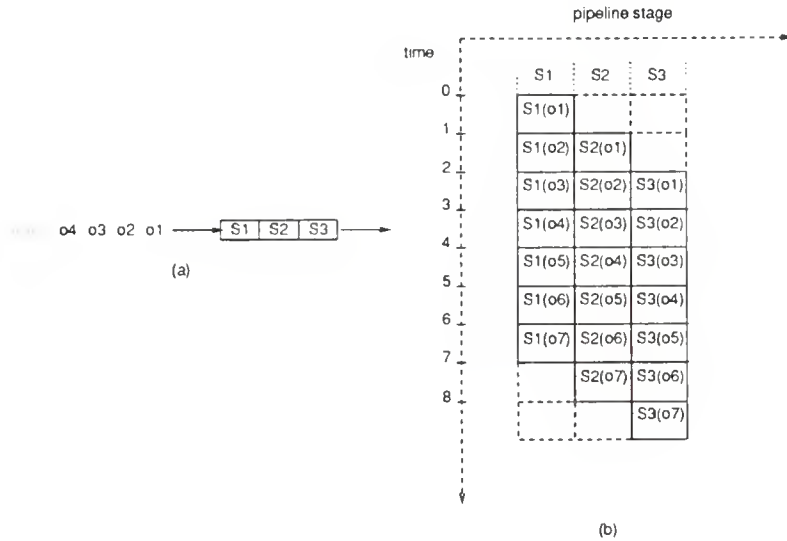


Figure 17: (a) A 3-stage hardware pipeline. (b) Time diagram of the hardware pipeline. Time elapses vertically, stages horizontally. Boxes are labeled $S_x(o_y)$ meaning that in a given time slot, stage S_x is executing operation o_y .

software pipeline (figure 18(b)), that is we initiate a new iteration every cycle, we obtain the time diagram of figure 18(c). Note the similarity with hardware pipelining. From the time diagram we see that the loop can be restructured so as to achieve the effect of a software pipeline. Figure 18(d) gives the software pipelined loop. The first two and last two instructions of the code are respectively called *loop prelude* (or *prolog*) and *loop postlude* (or *epilog*). Machine pipelines get filled during loop preludes, they are in their steady state while the pipelined loop body is executing, and they drain during loop postludes. Note that unrolling and compacting a loop does not have a similar effect, as machine pipelines usually drain every *unroll* iterations, where *unroll* is the amount of loop unrolling. Furthermore, software pipelining usually produces quite compact loop bodies, while loop unrolling does not. This can be a serious problem if the instruction cache is small.

It is not always possible to start an iteration every cycle, as iteration $i + 1$ might depend on iteration i . In the general case loop carried dependences (and resource availability) dictate the delay (called *initiation interval* [Rau81]) that one needs to respect before starting the next iteration. The goal of a compiler is to

minimize such delay. Software pipelining is more general than vectorization since all vectorizable loops can be software pipelined while the contrary does not hold. This is one important advantage of VLIW architectures over vector machines.

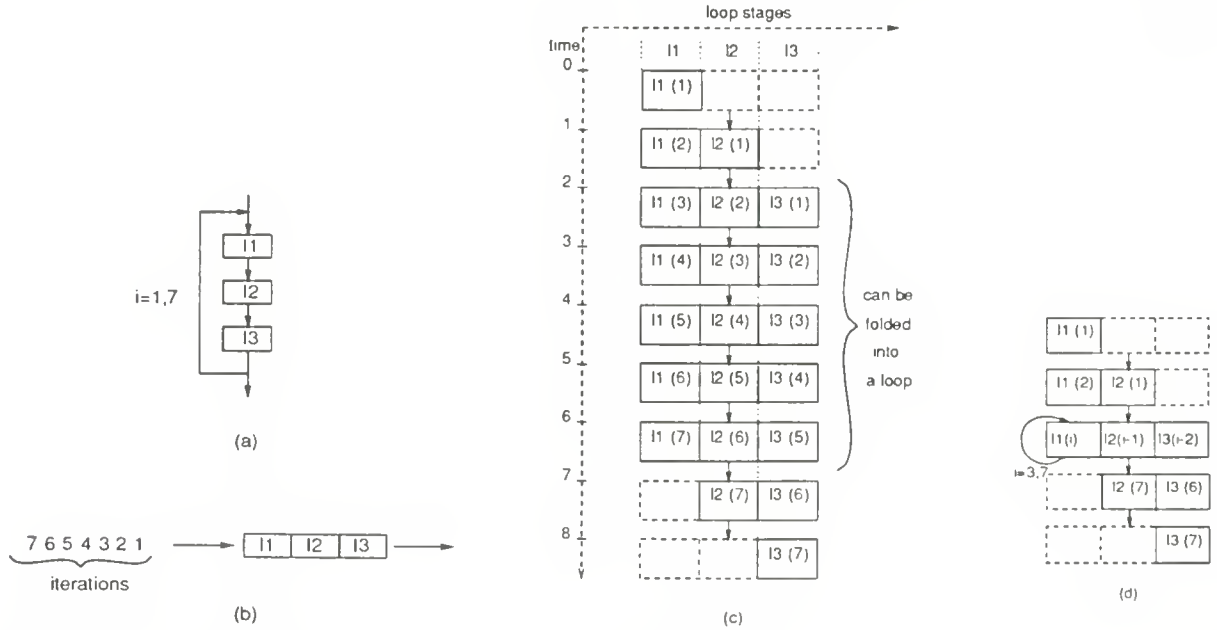


Figure 18: (a) A 3-instructions loop. Assume no loop carried dependences. (b) The loop viewed as a 3 stage software pipeline. Instruction I1 is stage 1, I2 stage 2, I3 stage 3. (c) Time diagram of the software pipeline. $I_x(y)$ means that at a given time the processor is executing the I_x th instruction of the y th iteration. (d) Software pipelined loop.

The loop pipelining idea is not new, Kogge [Kogge77] used it for handcoding microprogrammed pipelined machines, while recently Cytron has explored its application to multiprocessors [Cytron86]. Software pipelining algorithms can be divided into two classes:

1. straight line software pipelining techniques (section 5.1) pipeline loops with branch free bodies; they have mainly been developed for horizontal machines with limited parallelism;
2. complex software pipelining techniques (section 5.2), developed for highly parallel VLIWs, can pipeline loops containing arbitrary conditional branches.

In what follows loops are identified with their bodies and procedure calls are assumed to be inlined.

5.1 Straight Line Software Pipelining Techniques

The previous section has introduced software pipelining as being the software analog of hardware pipelining. While this analogy motivates the name, straight line software pipelining techniques view loop pipelining from a different (although equivalent) angle, based on the initiation interval concept. Consider a loop $\mathcal{L}$ comprising $l_{\mathcal{L}}$ instructions (unless otherwise stated, all loops considered in this and the following subsections have straight line bodies). When $\mathcal{L}$ is executed serially an iteration is started every $l_{\mathcal{L}}$ cycles (assuming an instruction per cycle execution speed). To speedup $\mathcal{L}$, successive iterations can be initiated every $t_{ii} < l_{\mathcal{L}}$ cycles (t_{ii} is the initiation interval)⁷. If we draw a time/iteration execution diagram of the loop in the two

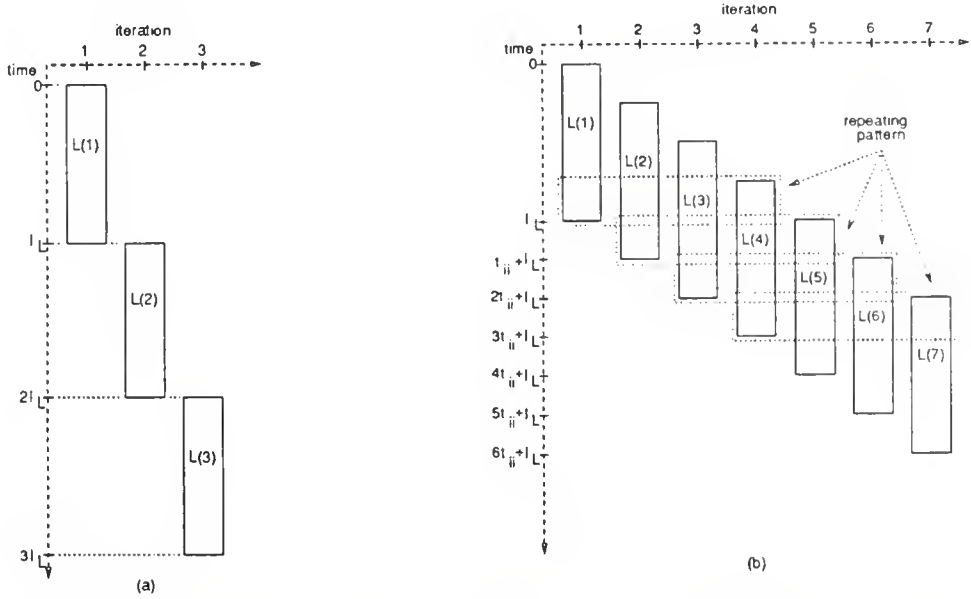


Figure 19: (a) Time/iteration execution diagram of $\mathcal{L}$ when $t_{ii} = l_{\mathcal{L}}$. (b) Time/iteration execution diagram of $\mathcal{L}$ when $t_{ii} < l_{\mathcal{L}}$.

cases $t_{ii} = l_{\mathcal{L}}$ and $t_{ii} < l_{\mathcal{L}}$ we obtain figures 19(a) and 19(b). If $\mathcal{L}$ has a sufficient number of iterations a repeating pattern can be detected. For instance consider a three instruction loop such as the one of figure 18(a), the repeating pattern is shown in figure 20(a), the software pipelined loop in 20(b).

A simple straight line software pipelining algorithm which generalizes the approach illustrated in figure 20 has been developed by Charlesworth for the FPS-164 [Charlesworth81]. The FPS-164 is an horizontal machine with modest parallelism, having one adder and one multiplier. Charlesworth algorithm is restricted to loops with known bounds and no carried dependences. The algorithm proceeds as follows:

1. $\mathcal{L}$ is compacted using some local compaction method like list scheduling. Let $\mathcal{L}_{lc}$ be the loop obtained after such compaction.

⁷As we will see later on the value of t_{ii} is tightly related to machine resource constraints and $\mathcal{L}$'s loop carried dependences.

2. Determine n_{cr} , the number of times the most frequently used resource (critical resource) is used in the loop. For the FPS $n_{cr} \leq t_{ii}$ (by the pigeonhole principle). Initially set t_{ii} to n_{cr} .
3. Let $\mathcal{L}_{sp}$ be the software pipelined loop. $\mathcal{L}_{sp}$ is generated from $\mathcal{L}_{lc}$ by packing every t_{ii} th instruction of $\mathcal{L}_{lc}$ into one $\mathcal{L}_{sp}$ instruction (array indices have to be adjusted). If resource or (intra-loop) data dependence conflicts arise, add one to t_{ii} , and repeat this step until the entire $\mathcal{L}_{lc}$ is folded into $\mathcal{L}_{sp}$.
4. Add to $\mathcal{L}_{sp}$ the loop exit test, code to increment array indices and generate the prelude and postlude to fill and empty the software pipeline.

Note that the body of $\mathcal{L}_{sp}$ has t_{ii} instructions. The value $\lceil l_{\mathcal{L}}/t_{ii} \rceil$, called *degree of pipelining* (and denoted Dp), is the number of iterations executed in parallel in the steady state and the number of times we need to iterate $\mathcal{L}_{sp}$ to execute one full $\mathcal{L}$ iteration. Dp is a close approximation of the speedup achieved by software pipelining. The previous algorithm assumes a constant initiation interval and a fixed schedule for each $\mathcal{L}$

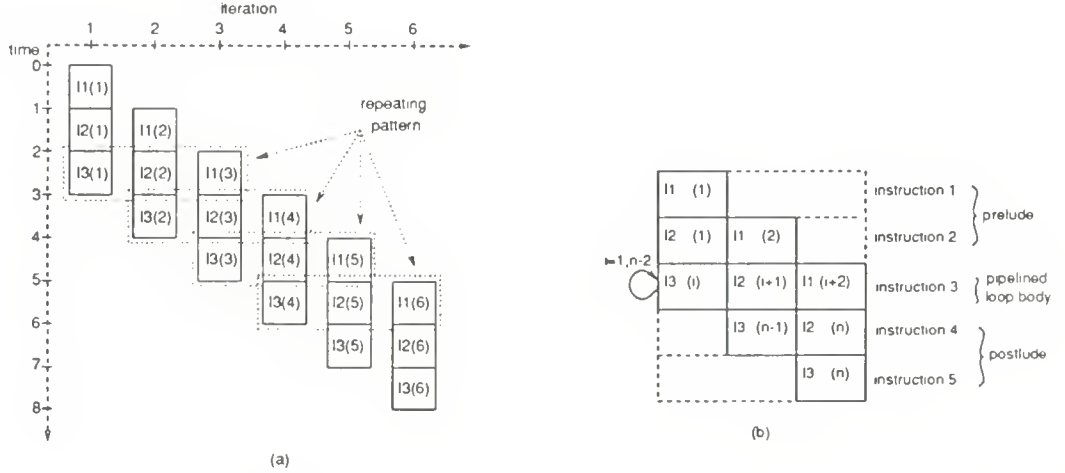


Figure 20: (a) Time/iteration execution diagram of the loop in figure 18. Note the repeating pattern $I3(i); I2(i+1); I1(i+2)$. (b) The software pipelined loop.

iteration, but this need not be. A variable initiation interval combined with variable iteration schedules might in fact yield a better throughput. Rau and Glaeser [Rau81] suggest to use a constant t_{ii} , and the same schedule for every iteration to reduce the size of the pipelined loop, and the complexity of software pipelining itself.

Let us illustrate Charlesworth method on the loop of figure 21(a), its intermediate representation is given in 21(b). The target machine is the FPS, we assume an execution rate of an instruction per cycle. $\mathcal{L}$ is compacted with bottom up list scheduling, the result (written in the PS formalism) is given in 21(c). The most critical resource of the loop is the multiplier, hence $n_{cr} = 2$. Initially $t_{ii} = 2$. We create $\mathcal{L}_{sp}$ from $\mathcal{L}_{lc}$ by packing every other instruction of $\mathcal{L}_{lc}$ into an $\mathcal{L}_{sp}$ instruction (array indices have to be adjusted), the result is given in 21(d). Of course in the final code $a(i+2)$, $b(i+2)$, $y(i+2)$, $z(i+2)$ have to be translated into

$i' := i + 2$ and $a(i')$, $b(i')$, $y(i')$, $z(i')$. After adding to $\mathcal{L}_{sp}$ the loop exit test, code to increment array indices, the prolog and epilog, we finally obtain the software pipelined loop of figure 21(e).

Charlesworth straight line software pipelining method is very simple. An early stage improvement over Charlesworth algorithm is the unroll pipeline reroll method (URPR) of Su [Su84,Su86,Mueller86] which can pipeline loops with carried dependences. Like Charlesworth technique URPR is limited to loops with known bounds. The idea behind URPR is as follows: if operation op_2 of iteration i (denoted $op_2(i)$) is data dependent on operation op_1 of iteration $i - 1$ (denoted $op_1(i - 1)$), then a delay equal to the execution time of op_1 must be respected between the execution of $op_1(i - 1)$ and the execution of $op_2(i)$. Such delay restricts the value of t_{ii} . The algorithm assumes an instruction per cycle execution rate, it proceeds as follows:

1. $\mathcal{L}$ is compacted using list scheduling. Let $\mathcal{L}_{lc}$ be the loop obtained after such compaction.
2. Compute n_{cr} , the number of times the critical resource is used in $\mathcal{L}_{lc}$.
3. Let $instruction\_num(op)$, be the number of the instruction to which an operation op of $\mathcal{L}_{lc}$ belongs. Consider the data dependence graph of $\mathcal{L}_{lc}$. To respect loop carried dependences from one $\mathcal{L}_{lc}$ iteration to the next a delay D of:

$$D = 1 + \max\{instruction\_num(op_2) - instruction\_num(op_1) \mid op_1, op_2 \in \mathcal{L}_{lc} \text{ and } \lambda(op_1, op_2) = 1\}$$

must be respected between the execution of successive $\mathcal{L}_{lc}$ iterations (λ was defined in section 2.1). Assuming a single instance for the critical resource, $t_{ii} = \max(n_{cr}, D)$.

4. **Unroll Phase.** Let $l_{\mathcal{L}_{lc}}$ be the number of instructions in $\mathcal{L}_{lc}$. Compute $Dp = \lceil l_{\mathcal{L}_{lc}} / t_{ii} \rceil$ the degree of pipelining. Unroll $\mathcal{L}_{lc}$ to create Dp loop bodies: $\mathcal{L}_{lc}(i), \mathcal{L}_{lc}(i + 1), \dots, \mathcal{L}_{lc}(i + Dp - 1)$.
5. **Pipeline Phase.** The Dp iterations are pipelined by placing them into a grid where rows represent machine cycles, and the columns contain the successive $\mathcal{L}_{lc}$ iterations. Iteration $\mathcal{L}_{lc}(i + k)$ is placed in column k starting at cycle $(k \cdot t_{ii})$. The schedule of $\mathcal{L}_{lc}(i + k)$ may be stretched to satisfy loop carried dependences with the previously scheduled $k - 1$ iterations (note that step 3 only takes into account loop carried dependences from one iteration to the next). If resource conflicts occur between an instruction I of $\mathcal{L}_{lc}$ scheduled in cycle c , and some instruction of a previously scheduled iteration, I is placed in cycle $c + 1$. If a resource conflict still occurs, a new empty row is created between rows c and $c + 1$ where I is inserted. The above process is repeated until all the $\mathcal{L}_{lc}(i + k)$ s have been scheduled.
6. **Reroll Phase.** The Dp iterations are rerolled into a new loop body which includes all the instructions of $\mathcal{L}_{lc}$, and is of minimal length. Such loop body is determined by exhaustive search. After adding the exit test, and generating the loop prelude and postlude we finally obtain the software pipelined loop.

As an example consider a loop with no carried dependences whose compacted loop $\mathcal{L}_{lc}$ has the resource requirements shown in figure 22(a) (+, $\times$, $\div$, $\sqrt{\quad}$, respectively mean floating point addition, multiplication, division and square root). Since there are no loop carried dependences and $n_{cr} = 3$, $t_{ii} = 3$. We need to pipeline $Dp = \lceil 7/3 \rceil = 3$ consecutive iterations. The result of the pipelining phase is given in 22(b). After rerolling we obtain the loop body given in 22(c).

FOR i IN 1..100 LOOP	t1 := a(i)	$L_1: [\{t1:=a(i); t2:=y(i)\}, \phi, L_2]$
x(i) := a(i)*y(i)+b(i)*z(i);	t2 := y(i)	$L_2: [\{t3:=t1*t2; t4:=b(i); t5:=z(i)\}, \phi, L_3]$
END LOOP;	t3 := t1*t2	$L_3: [\{t6:=t4*t5\}, \phi, L_4]$
	t4 := b(i)	$L_4: [\{t7:=t3+t6\}, \phi, L_5]$
	t5 := z(i)	$L_5: [\{x(i):=t7\}, \phi, —]$
	t6 := t4*t5	
	t7 := t3+t6	
	x(i) := t7	

(a)
(b)
(c)

$L_a: [\{x(i):=t7; t6:=t4*t5; t1:=a(i+2); t2:=y(i+2)\}, \phi, L_b]$
 $L_b: [\{t7:=t3+t6; t3:=t1*t2; t4:=b(i+2); t5:=z(i+2)\}, \phi, —]$
 (d)

$P_1: [\{t1:=a(1); t2:=y(1)\}, \phi, P_2]$
 $P_2: [\{t3:=t1*t2; t4:=b(1); t5:=z(1)\}, \phi, P_3]$
 $P_3: [\{t6:=t4*t5; t1:=a(2); t2:=y(2)\}, \phi, P_4]$
 $P_4: [\{t7:=t3+t6; t3:=t1*t2; t4:=b(2); t5:=z(2); i:=1; i':=3\}, \phi, L_a]$

 $L_a: [\{x(i):=t7; t6:=t4*t5; t1:=a(i'); t2:=y(i'); c:=(i>97)\}, \phi, L_b]$
 $L_b: [\{t7:=t3+t6; t3:=t1*t2; t4:=b(i'); t5:=z(i'); i:=i+1; i':=i'+1\}, \{IF c THEN E_1 ELSE L_a\}, —]$

 $E_1: [\{x(99):=t7; t6:=t4*t5\}, \phi, E_2]$
 $E_2: [\{t7:=t3+t6\}, \phi, E_3]$
 $E_3: [\{x(100):=t7\}, \phi, Exit]$

Exit:

 (e)

Figure 21: (a) The loop to be pipelined. (b) The intermediate code representation of the loop's body. (c) The loop after local compaction (bottom-up list scheduling). (d) The body of the software pipelined loop. (e) The final software pipelined loop.

Although URPR handles loop carried dependences, it does so in a very peculiar fashion as the value of t_{ii} only reflects those dependences which are carried from one iteration to the next. Also resource constraints are satisfied by stretching *fixed* iteration schedules which are imposed by the local compaction phase of URPR. This appears to be somewhat awkward. Generalized URPR (GURPR) [Su87] extends URPR to handle practically any kind of loop. A list of techniques is given to patch URPR in the event of conditional branches, procedure calls, etc. It however remains that general loop carried dependences are not adequately handled, as GURPR is built on top of URPR.

<i>cycle</i>	$\mathcal{L}_{lc}$
0	$\sqrt{\quad}$
1	+
2	$\times \div$
3	$\times$
4	+
5	$\times \div$
6	$\times$

(a)

<i>cycle</i>	$\mathcal{L}_{lc}(i)$	$\mathcal{L}_{lc}(i+1)$	$\mathcal{L}_{lc}(i+2)$
0	$\sqrt{\quad}$		
1	+		
2	$\times \div$		
3	$\times$	$\sqrt{\quad}$	
4	+	no-op	
5	no-op	+	
6	$\times \div$	no-op	$\sqrt{\quad}$
7	no-op	$\times \div$	+
8	$\times$	no-op	no-op
9		no-op	$\times \div$
10		$\times$	no-op
11		+	$\times$
12		$\times \div$	no-op
13		$\times$	+
14			$\times \div$
15			$\times$

(b)

<i>cycle</i>	<i>rerolled body</i>	
0	$\times$	$\sqrt{\quad}$
1	+	no-op
2	no-op	+
3	$\times \div$	no-op
4	no-op	$\times \div$
5	$\times$	no-op

(c)

Figure 22: (a) Resource requirements of some loop after compaction with list scheduling. (b) The loop after the pipelining phase. (c) The rerolled loop body.

The pipelining algorithms of Charlesworth and Su illustrate the initial attempts to solve the straight line software pipelining problem (formally defined in section 5.1.1). Another early attempt was made by Touzeau [Touzeau84]. His method adequately pipelines loops with arbitrary carried dependences, but can only process *single* statement loops. The algorithm is similar to the more general technique of Lam presented in section 5.1.2: given an initiation interval, Touzeau's method tries to rearrange the operations of the loop processed such that resource and data dependence constraints are satisfied. The smallest initiation interval for which a valid loop schedule can be unveiled is determined by binary search.

After formally defining the straight line software pipelining problem in section 5.1.1 we present two recent straight line software pipelining algorithms developed by Lam [Lam88] (section 5.1.2) and Aiken and Nicolau [Aiken88b] (section 5.1.4), which give optimal or quasi-optimal results.

5.1.1 Formal Definition of the Straight Line Software Pipelining Problem

Let $t_{ii}(i, i+1)$ be the initiation interval between iterations i and $i+1$ ⁸, and let $t_{ii}(i, i+k)$ be the initiation interval between iterations i and $i+k$:

$$t_{ii}(i, i+k) = \sum_{h=0}^{k-1} t_{ii}(i+h, i+h+1)$$

The most general definition of the straight line software pipelining problem (SLSP problem) is: **given**

1. a machine $\mathcal{M}$ with a set of resource constraints,
2. a loop $\mathcal{L}$ comprising n iterations
3. $\mathcal{L}$'s data dependence graph,

the **goal** is to determine schedules $\sigma_1, \sigma_2, \dots, \sigma_i, \dots, \sigma_n$ and initiation intervals $t_{ii}(1, 2), \dots, t_{ii}(i-1, i), \dots, t_{ii}(n-1, n)$ respectively for iterations $1, 2, \dots, i, \dots, n$ such that $\mathcal{M}$ resource constraints and $\mathcal{L}$ data dependences are satisfied, and $\mathcal{L}$ executes in the shortest possible time, i.e.

$$\max_{1 \leq i \leq n} (t_{ii}(1, i) + \text{len}(\sigma_i)) \text{ is minimum}$$

($\text{len}(\sigma_i)$ is the length of schedule σ_i (see section 3.1 for notations))⁹.

According to the above definition every iteration could potentially have a different schedule, this would entail very poor space efficiency since the code for each iteration would have to be fully specified. This has led researchers like Rau [Rau81] and Charlesworth [Charlesworth81] (see previous section) to define a restricted version of the general SLSP problem which imposes $\sigma_1 = \sigma_2 = \dots = \sigma_i = \dots = \sigma_n = \sigma$ and $t_{ii}(1, 2) = \dots = t_{ii}(i-1, i) = \dots = t_{ii}(n-1, n) = t_{ii}$. The goal of the restricted SLSP problem is to find the shortest loop schedule σ and the minimum *constant* initiation interval t_{ii} , such that $\mathcal{M}$ resource constraints and $\mathcal{L}$ data dependences are respected when a new iteration is started every t_{ii} cycles. Given a loop schedule σ and a constant initiation interval t_{ii} that satisfy resource and data dependence requirements, extremely compact pipelined loop code can be generated (like in Charlesworth algorithm) by packing together every t_{ii} th instruction of σ .

Unfortunately while optimal solutions to the general SLSP problem guarantee time optimal results, optimal solutions to the restricted SLSP problem do not. To see this consider the loop of figure 23(a), whose data dependence graph is given in 23(b). If we assume a machine with infinite resources and a 1 cycle delay for all operations, the optimal solution to the restricted SLSP problem is given in 23(c). The solution is not time optimal, it takes $2 \cdot n$ cycles to execute the whole loop. The optimal solution to the general SLSP problem is given in 23(d), it requires $\lceil \frac{3n}{2} \rceil$ time steps.

We now formally define the general and restricted SLSP problems. The input to the two problems is similar to the local compaction problem (LCP) input, except for the fact that LCP deals with acyclic instead of cyclic code (refer to section 3.1 for notations).

⁸The initiation interval $t_{ii}(i, i+1)$ is the delay that separates the beginning of execution of iterations i and $i+1$.

⁹For all iteration i ($1 \leq i \leq n$) and for all operation op of $\mathcal{L}$, $\sigma_i(op)$ gives the *relative* schedule time of op within iteration i . The *absolute* schedule time of op during iteration i is $t_{ii}(1, i) + \sigma_i(op)$.

```

FOR i IN 1..n LOOP
  a(i) := c(i-1);  -- A
  b(i+1) := a(i);  -- B
  c(i) := b(i);    -- C
END LOOP;

```



(a)

(b)

$$t_{ii} = 2$$

σ	
0	AC
1	B

cycle	iterations			
	1	2	3	...
0	AC			
1	B			
2		AC		
3		B		
4			AC	
5			B	

(c)

$$t_{ii}(i, i+1) = \begin{cases} 1 & \text{if } i \text{ is odd} \\ 2 & \text{if } i \text{ is even} \end{cases}$$

$$\sigma_i = \begin{cases} \begin{array}{|c|c|} \hline \sigma_i & \\ \hline 0 & AC \\ \hline 1 & B \\ \hline \end{array} & \text{if } i \text{ is odd} \\ \begin{array}{|c|c|} \hline \sigma_i & \\ \hline 0 & A \\ \hline 1 & BC \\ \hline \end{array} & \text{if } i \text{ is even} \end{cases}$$

cycle	iterations				
	1	2	3	4	...
0	AC				
1	B	A			
2		BC			
3			AC		
4			B	A	
5				BC	

(d)

Figure 23: (a) A loop. (b) Its data dependence graph. Dashed edges indicate carried dependences from one iteration to the next ($\lambda(e) = 1$). (c) Optimal solution to the restricted SLSP problem. t_{ii} and σ are given to the right, the time/iteration execution diagram is given to the left. (d) Optimal solution to the general SLSP problem. The t_{ii} s and σ_i s are given to the right, the time/iteration execution diagram is given to the left.

Given

1. a machine $\mathcal{M}$, a set of resources $\mathcal{R} = \{r_1, \dots, r_m\}$, and a resource configuration vector $\vec{R}_{\mathcal{M}}$,
2. a loop $\mathcal{L}$ comprising n iterations and composed of operations $\mathcal{O} = \{op_1, \dots, op_j, \dots, op_l\}$,
3. a duration function $d : \mathcal{O} \rightarrow \mathbb{N}$ and a resource usage function $\vec{R}_{\mathcal{O}} : \mathcal{O} \times \mathbb{Z} \rightarrow \mathbb{N}^m$,
4. a data dependence graph $\text{DDG} = (\mathcal{O}, E)$ imposing a partial order on $\mathcal{O}$
5. a tuple of functions $[\lambda(e), \delta(e)]$ associated with each edge e of the DDG (remember from section 2.1 that each edge $e = (op_{j_1}, op_{j_2}) \in E$ has associated a tuple $[\lambda(e), \delta(e)]$, whose meaning is: operation op_{j_2} must execute $\delta(e)$ cycles after operation op_{j_1} of the $\lambda(e)$ th previous iteration has started executing);

the goal is to find

General problem: schedules $\sigma_1, \sigma_2, \dots, \sigma_i, \dots, \sigma_n : \mathcal{O} \rightarrow \mathbb{N}$ and initiation intervals

$t_{ii}(1, 2), \dots, t_{ii}(i-1, i), \dots, t_{ii}(n-1, n)$ respectively for iterations $1, 2, \dots, i, \dots, n$

Restricted problem: a fixed loop schedule $\sigma : \mathcal{O} \rightarrow \mathbb{N}$ and a constant initiation interval t_{ii}

subject to the following conditions:

1. minimality:

General problem: $\max_{1 \leq i \leq n} (t_{ii}(1, i) + \text{len}(\sigma_i))$ is minimum

Restricted problem: t_{ii} and $\text{len}(\sigma)$ are minimum

2. dependence constraint:

General problem: given a data dependence edge $e = (op_{j_1}, op_{j_2}) \in E$ there are two cases to consider:

(a) if $\lambda(e) = 0$ then we must have: $\forall 1 \leq i \leq n \quad \sigma_i(op_{j_2}) - \sigma_i(op_{j_1}) \geq \delta(e)$

(b) if $\lambda(e) \neq 0$ then we must have:

$$\forall 1 \leq i \leq n - \lambda(e) \quad t_{ii}(i, i + \lambda(e)) + \sigma_{i+\lambda(e)}(op_{j_2}) - \sigma_i(op_{j_1}) \geq \delta(e)$$

By merging the two conditions we obtain:

$$\forall e = (op_{j_1}, op_{j_2}) \in E \quad \forall 1 \leq i \leq n - \lambda(e) \quad \sigma_{i+\lambda(e)}(op_{j_2}) - \sigma_i(op_{j_1}) \geq \delta(e) - t_{ii}(i, i + \lambda(e))$$

Restricted problem: in the restricted case we have $\sigma_{i+\lambda(e)} = \sigma_i = \sigma$ and $t_{ii}(i, i + \lambda(e)) = \lambda(e) \cdot t_{ii}$, this gives the following data dependence constraint:

$$\forall e = (op_{j_1}, op_{j_2}) \in E \quad \sigma(op_{j_2}) - \sigma(op_{j_1}) \geq \delta(e) - \lambda(e) \cdot t_{ii}$$

3. resource constraint:

General problem: extending the resource constraint of section 3.1 to the iterative case gives:

$$\forall 0 \leq t < \max_{1 \leq i \leq n} (t_{ii} + \text{len}(\sigma_i)) \quad \sum_{j=1}^l \sum_{i=1}^n \vec{R}_{\mathcal{O}}(op_j, t - (t_{ii}(1, i) + \sigma_i(op_j))) \preceq \vec{R}_{\mathcal{M}}$$

Restricted problem: the resource constraint for the restricted problem can be derived from the previous formula. However a more intuitive understanding is obtained by deriving

the formula directly. In the restricted case iterations all have the same schedule and are initiated at the rate of one iteration every t_{ii} cycles, the total resource requirements of every t_{ii} th instruction in $\mathcal{L}$ should therefore not exceed the available limit (note that operations lasting more than t_{ii} cycles must be handled properly). All these considerations yield the following constraint:

$$\forall 0 \leq t < t_{ii} \quad \sum_{j=1}^l \sum_{x=0}^{\lfloor d(op_j)/t_{ii} \rfloor} \vec{R}_O(op_j, t - (\sigma_i(op) \bmod t_{ii}) + x \cdot t_{ii}) \preceq \vec{R}_M$$

Like the local compaction problem, the restricted and general SLSP problems are NP-complete [Lam87,Garey79].

Lam's algorithm (section 5.1.2) is based on the restricted SLSP problem. As we already mentioned this choice is motivated by the concern of minimizing object code size, and by the fact that in the infinite resource case (no resource constraints) optimal results for the restricted SLSP problem are guaranteed never to be more than n cycles slower than the time optimum (see section 5.1.2). Aiken and Nicolau's algorithm (section 5.1.4) gives an optimal solution to the general SLSP problem in the infinite resource case.

5.1.2 A General Algorithm for the Restricted SLSP Problem

The pipelining techniques of Charlesworth and Su are too rigid: the initially fixed loop schedule imposes an unnecessary constraint on the initiation interval t_{ii} . One of the reasons for this rigidity comes from the circular dependency between t_{ii} and σ (scheduling constraints are defined in terms of the initiation interval t_{ii} , however determining if t_{ii} satisfies resource and data dependence constraints depends on the schedule σ chosen).

To resolve this circularity Lam [Lam87,Lam88] adopts an iterative approach (see below). Her method can handle any type of straight line loop body. The algorithm proceeds as follows:

1. Based on resource and data dependence constraints, determine the lower and upper bound of t_{ii} . Let t_{min} and t_{max} be such lower and upper bounds. Initially set t_{ii} to t_{min} .
2. Partition the loop DDG into strongly connected components.
3. Find a separate schedule for each strong component using an adapted list scheduling algorithm.
4. Condense each strong component into a single node.
5. Schedule the condensed DDG (which is now acyclic) using list scheduling.
6. If any of steps 3 or 5 fails (i.e. we cannot find a schedule given the current t_{ii}), increment t_{ii} by one and repeat steps 3 through 5. If t_{ii} is set to t_{max} the loop cannot be pipelined.

Let $\mathcal{L}$ be the loop to be pipelined. The detailed description of the algorithm follows.

Lower and upper bounds on t_{ii} . The upper bound t_{max} is easily determined by applying list scheduling to $\mathcal{L}$: t_{max} is the length of the schedule obtained from list scheduling. The lower bound t_{min} is imposed by resource and dependence constraints.

1. **Resource constraints:** the final software pipelined loop $\mathcal{L}_{sp}$ contains all the operations in $\mathcal{L}$. Since $\mathcal{L}_{sp}$ takes t_{ii} cycles to execute, the total number of resource units available in t_{ii} cycles must be greater than or equal to the resource requirements of all $\mathcal{L}$'s operations put together (this is the critical resource constraint of Charlesworth adapted to the multiple resource case), that is:

$$\begin{aligned} \forall 1 \leq k \leq m \quad t_{ii} \cdot \bar{R}_{\mathcal{M}}(k) &\geq \sum_{j=1}^l \sum_{x=0}^{d(op_j)-1} \bar{R}_{\mathcal{O}}(op_j, x)(k) \\ \Rightarrow t_{ii} &\geq \max_{1 \leq k \leq m} \frac{\sum_{j=1}^l \sum_{x=0}^{d(op_j)-1} \bar{R}_{\mathcal{O}}(op_j, x)(k)}{\bar{R}_{\mathcal{M}}(k)} = t_{min_R} \end{aligned}$$

2. **Dependence constraints:** cycles in the DDG impose a lower bound on the initiation interval. Let $\mathcal{C} = [op_{j_0}, op_{j_1}, \dots, op_{j_{\epsilon-1}}]$ be a cycle of the DDG. For each edge $e = (op_{j_r}, op_{j_{(r+1) \bmod \epsilon}})$ the dependence constraint imposes $\sigma(op_{j_{(r+1) \bmod \epsilon}}) - \sigma(op_{j_r}) \geq \delta(e) - t_{ii} \cdot \lambda(e)$, that is if we sum all the $\mathcal{C}$ inequalities together,

$$t_{ii} \geq \delta(\mathcal{C}) / \lambda(\mathcal{C})$$

where $\delta(\mathcal{C})$ and $\lambda(\mathcal{C})$ are the sum of the individual δ s and λ s of each edge in $\mathcal{C}$. Thus

$$t_{ii} \geq \max_{\mathcal{C} \in \text{cycles of DDG}} [\delta(\mathcal{C}) / \lambda(\mathcal{C})] = t_{min_D}$$

Taking the maximum of t_{min_R} and t_{min_D} gives $t_{min} = \max(t_{min_R}, t_{min_D})$.

Partitioning the DDG. The DDG is partitioned into strong components by using any of the standard algorithms [Tarjan72, Sharir81].

Scheduling a strong component. Let C be a strong component of the DDG, and let op_{j_1}, op_{j_2} be any two operations in C . Assume op_{j_1} has been scheduled, that is $\sigma(op_{j_1})$ is known. Adding dependence constraints along $LP_{op_{j_1} \rightarrow op_{j_2}}$ and $LP_{op_{j_2} \rightarrow op_{j_1}}$ (the longest path from op_{j_1} to op_{j_2} , and from op_{j_2} to op_{j_1})¹⁰ yields:

$$\sigma(op_{j_1}) - (\delta(LP_{op_{j_2} \rightarrow op_{j_1}}) - t_{ii} \cdot \lambda(LP_{op_{j_2} \rightarrow op_{j_1}})) \geq \sigma(op_{j_2}) \geq \sigma(op_{j_1}) + \delta(LP_{op_{j_1} \rightarrow op_{j_2}}) - t_{ii} \cdot \lambda(LP_{op_{j_1} \rightarrow op_{j_2}})$$

thus the scheduling interval of the unscheduled C operations shrinks every time a new operation of C is scheduled.

C operations are scheduled by applying list scheduling to the subgraph composed of all the edges e of C such that $\lambda(e) = 0$ (such subgraph is acyclic). The priority function used is *earliest deadline first*. Here is how the scheduling algorithm proceeds.

1. Let **Data_Ready** be the set of operations of C whose incoming edges have non zero λ value.

¹⁰To determine longest paths the "all-pairs longest path problem" is solved with the cost of an edge e set to $\delta(e) - t_{ii} \cdot \lambda(e)$.

2. Pick any operation op_{j_o} in `Data_Ready` and set $\sigma(op_{j_o}) = 0$. Then for each unscheduled op_{j_r} , compute $[\sigma_{min}(op_{j_r}); \sigma_{max}(op_{j_r})]$, the time interval in which op_{j_r} has to be scheduled ($\sigma_{min}(op_{j_r})$ and $\sigma_{max}(op_{j_r})$ are computed using the inequalities of the previous paragraph):

$$\begin{aligned}\sigma_{min}(op_{j_r}) &= \sigma(op_{j_o}) + \delta(LP_{op_{j_o} \rightarrow op_{j_r}}) - t_{ii} \cdot \lambda(LP_{op_{j_o} \rightarrow op_{j_r}}) \\ \sigma_{max}(op_{j_r}) &= \sigma(op_{j_o}) - (\delta(LP_{op_{j_r} \rightarrow op_{j_o}}) - t_{ii} \cdot \lambda(LP_{op_{j_r} \rightarrow op_{j_o}}))\end{aligned}$$

3. Put all the operations whose operands have been scheduled in `Data_Ready`.
4. Select the operation $op_{j_s} \in \text{Data\_Ready}$ such that $\sigma_{max}(op_{j_s}) = (\min_{op \in \text{Data\_Ready}} \sigma_{max}(op))$, i.e. the operation with the earliest deadline.
5. Schedule op_{j_s} by scanning all the time slots from $\sigma_{min}(op_{j_s})$ to $\sigma_{max}(op_{j_s})$, selecting the first cycle in which the restricted resource constraints are satisfied. If such time slot does not exist stop: a schedule for C cannot be found with the current t_{ii} (t_{ii} has to be increased). Assuming a time slot is found, set $\sigma(op_{j_s})$ to that time slot.
6. Recompute σ_{min} and σ_{max} for any unscheduled operation op_{j_r} :

$$\begin{aligned}\sigma_{min}(op_{j_r}) &= \max(\sigma_{min}(op_{j_r}), \sigma(op_{j_s}) + \delta(LP_{op_{j_s} \rightarrow op_{j_r}}) - t_{ii} \cdot \lambda(LP_{op_{j_s} \rightarrow op_{j_r}})) \\ \sigma_{max}(op_{j_r}) &= \min(\sigma_{max}(op_{j_r}), \sigma(op_{j_s}) - (\delta(LP_{op_{j_r} \rightarrow op_{j_s}}) - t_{ii} \cdot \lambda(LP_{op_{j_r} \rightarrow op_{j_s}})))\end{aligned}$$

7. Repeat steps 3 to 6 until all the operations of C have been scheduled or some operation cannot be scheduled.

Condensing strong components. Once a schedule σ_C for every strong component $C = (\mathcal{O}_C, E_C)$ has been determined, C is condensed into a single node in the DDG and is subsequently treated like any standard operation. The duration function of C is defined to be $d(C) = len(\sigma_C)$, its resource usage function is defined to be the sum of the resource usage functions of the operations that make up C , that is:

$$\bar{R}_C(C, x) = \begin{cases} \sum_{op_j \in \mathcal{O}_C} \bar{R}_{\mathcal{O}}(op_j, x - \sigma_C(op_j)) & \text{if } 0 \leq x < d(C) \\ \vec{0} & \text{otherwise} \end{cases}$$

The condensed DDG is obtained from the original DDG as follows. Let $e = (op_1, op_2)$ be an edge of DDG such that $op_1 \in C_1$ and $op_2 \in C_2$, C_1 and C_2 being two different strong components. Create an edge e' from C_1 to C_2 . Set $\lambda(e') = \lambda(e)$, and $\delta(e') = \delta(e) + (\sigma_{C_1}(op_1) - \sigma_{C_2}(op_2))$. If multiple edges between C_1 and C_2 arise, select the edge e' with biggest $\delta(e') - t_{ii} \cdot \lambda(e')$.

Scheduling strong components. The nodes of the condensed DDG (the strong components) are scheduled using list scheduling, with the exception that the resource constraint equation for the restricted SLSP problem is used instead of the one given in section 3.1. The priority function used takes into account the fact that nodes represent several operations: if C_1 represents a operations and C_2 represents b operations, $a > b$, C_1 is given priority over C_2 .

As we already mentioned in section 5.1.1, the body of the software pipelined loop can be obtained by packing together every t_{ii} th instruction of σ . Generating the prelude and postlude is straightforward if we assume that n , the number of iterations executed, is greater than or equal to $\lceil \text{len}(\sigma)/t_{ii} \rceil$ (that is we execute the body of the software pipelined loop at least once). To insure that it is always the case, a test performed at run time decides whether to execute the original loop $\mathcal{L}$ or the software pipelined loop $\mathcal{L}_{sp}$.

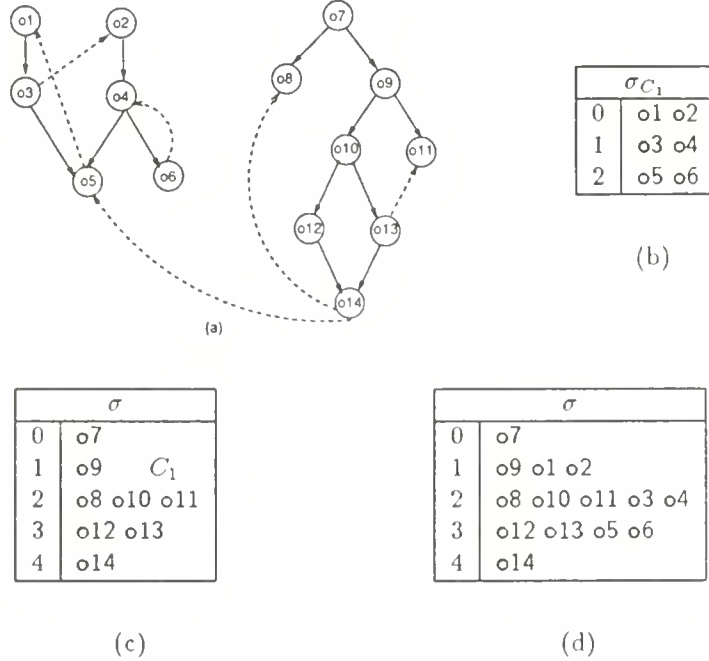


Figure 24: (a) Data dependence graph of some loop $\mathcal{L}$; all loop carried dependences (dashed edges) are from one iteration to the next ($\lambda(\epsilon) = 1$). (b) The schedule of strong component C_1 . (c) The loop schedule after scheduling the condensed DDG. (d) Final loop schedule (the initiation interval t_{ii} is 3).

We illustrate Lam's algorithm on a simple example. Assume a machine with unlimited resources (no resource constraints), where all operations take 1 cycle. Consider a loop with a data dependence graph as portrayed in figure 24(a). The DDG has nine strong components $C_1 = \{o1, \dots, o6\}$, $C_2 = \{o7\}$, $\dots$, $C_9 = \{o14\}$. The cycle $o1 \rightarrow o3 \rightarrow o5 \rightarrow o1$ of C_1 imposes $t_{ii} \geq 3$ ($t_{ii} = 3$ is the final initiation interval). The schedule obtained for C_1 is given in figure 24(b); after condensing C_1 and scheduling the condensed DDG with list scheduling we obtain the schedule of figure 24(c) which, once C_1 has been replaced by its schedule σ_{C_1} , yields the final schedule of figure 24(d).

Lam's algorithm might unnecessarily lengthen the value of t_{ii} if a variable is redefined at the beginning of every iteration. *Modulo variable expansion*, a variation of the scalar expansion technique of vectorizing compilers [Padua86], is introduced to avoid this problem. The pipelining algorithm is modified as follows:

1. Pretend that every iteration of $\mathcal{L}$ has a dedicated set of registers (so that no spurious loop carried output dependences arise because of temporaries).

2. Identify the variables which are redefined at the beginning of every iteration. “Expand those variables into arrays”, i.e. for each redefined variable use a different register in each iteration (this is the standard scalar expansion technique).
3. The algorithm then proceeds as previously stated. The resulting schedule σ is used to determine the actual number of registers that must be allocated to each redefined variable (and all the temporaries): if the lifetime of a register variable is lt cycles, then $\lceil lt/t_{ii} \rceil$ different values must be kept alive concurrently in that many registers. If a variable is allocated r registers, then the i th iteration will use the $(i \bmod r)$ th register allocated to the variable.
4. Without modulo variable expansion the number of VLIW instructions in the body of $\mathcal{L}_{sp}$ is t_{ii} . When modulo variable expansion is used, iterations using different sets of registers must execute different sequences of instructions. Let r_{v_i} be the number of registers allocated to variable v_i , and let u be the least common multiple of the r_{v_i} s. The body of the final software pipelined loop is obtained by packing together every t_{ii} th instruction of σ and then replicating u times the t_{ii} instructions obtained by the packing process (register variables have to be appropriately renamed during the replication). The final body of the software pipelined loop contains $u \cdot t_{ii}$ instructions.

Lam has also presented a simple hierarchical method to handle loops with conditional constructs, but with reducible flow graph. The THEN and ELSE branches of a conditional statement are first independently compacted using list scheduling, and then padded to the same length. Once this is done the entire conditional statement is reduced to a single node, much in the same way a strong component is condensed. This node can be treated within the surrounding construct (the loop or another conditional statement) like any other simple operation. At code emission time any operation scheduled concurrently with the conditional node is duplicated in both branches (note that this might lead to code explosion if conditional branches are scheduled concurrently). This method is very simple, and its main intent is to prevent conditionals from preventing loop pipelining. It remains, however, that the presence of conditionals inside a loop inhibits the effectiveness of software pipelining.

Lam’s software pipelining technique has been extensively tested on signal processing and scientific code for the Warp machine (a linear systolic array composed of 10 VLIW cells [Annaratone87]). To study the gain in performance obtained from software pipelining, Lam measured the speedup of pipelined versus compacted loops: the average factor of increase in speed was three.

In the infinite resource case (no resource constraints) Lam’s algorithm yields an optimal solution to the restricted SLSP problem. This optimal solution is never “too far” from the time optimum. We now formalize the “never too far” notion. Lam’s algorithm employs the data dependence cycles of $\mathcal{L}$ to establish a lower bound on t_{ii} . Strictly speaking however, cycles impose a scheduling constraint only on the operations of the cycle. Let $C = [op_{j_0}, op_{j_1}, \dots, op_{j_{c-1}}]$ be a data dependence cycle of $\mathcal{L}$. For each edge $e = (op_{j_r}, op_{j_{(r+1) \bmod c}})$ the general dependence constraint imposes:

$$\sigma_{i+\lambda(e)}(op_{j_{(r+1) \bmod c}}) - \sigma_i(op_{j_r}) \geq \delta(e) - t_{ii}(i, i + \lambda(e))$$

that is if we sum all the C inequalities together:

$$\forall 1 \leq i \leq n - \lambda(C) \quad \forall op_{j_r} \in C \quad \sigma_{i+\lambda(C)}(op_{j_r}) - \sigma_i(op_{j_r}) \geq \delta(C) - t_{ii}(i, i + \lambda(C))$$

where $\delta(C)$ and $\lambda(C)$ are the sum of the individual δ s and λ s of each edge in C . Lam's algorithm satisfies such constraints by scheduling consecutive copies of op_{j_r} at constant intervals of $t_{ii} = \lceil \delta(C)/\lambda(C) \rceil$ cycles. If $\lambda(C)$ divides $\delta(C)$ this yields a time optimal solution, otherwise it does not and the solution obtained is off the time optimum by:

$$(\lceil \frac{\delta(C)}{\lambda(C)} \rceil - \frac{\delta(C)}{\lambda(C)}) \cdot n < n \text{ cycles}$$

where n is the overall number of iterations executed.

5.1.3 An Optimal Algorithm for the Restricted SLSP Problem Using Special Hardware

The polycyclic architecture, developed by Rau and Glaeser [Rau81,Rau82a,Rau82b]¹¹, can be employed in conjunction with Lam's algorithm to obtain optimal solutions for the restricted SLSP problem in the case of loops with no carried dependences. The key idea of the architecture is to ensure that explicitly scheduled resources perform *only* those operations that are explicit in the loop data dependence graph. This is achieved by fully interconnecting functional units through a crossbar and placing a special dedicated buffer, called *delay element*, between the output and the inputs of every functional unit (see figure 25). A delay element

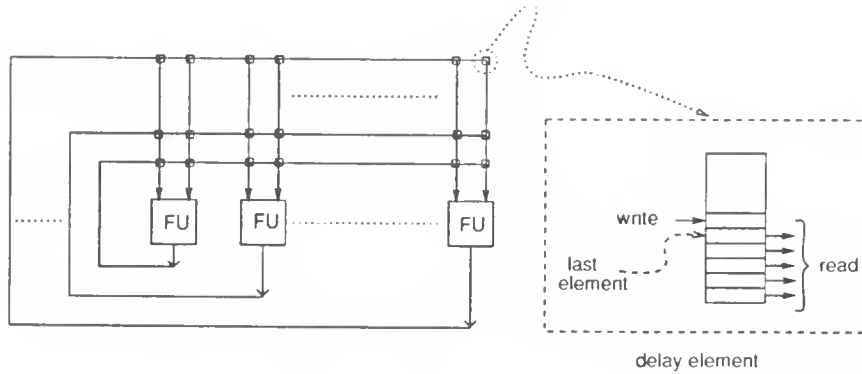


Figure 25: The output of every functional unit is connected to the inputs of every functional unit through a delay element.

is partly a queue and partly a register file i.e., data can be read from any location in the delay element, but can only be written at its end. Also, a read operation has the option of deleting the data read from the delay element by shifting down all the values above it.

Operations executed on the polycyclic architecture do not reference any registers, they just specify the functional units on which they execute, the location of the operands in the input delay elements, and the delay elements in which the results should be stored. No register allocation is therefore needed and the buffered crossbar has the effect of implicitly implementing the modulo variable expansion optimization of Lam. As we already mentioned optimal solutions for the restricted SLSP problem can be obtained by adapting Lam's algorithm to the polycyclic architecture. Note that optimal solutions can be obtained only for loops with no

¹¹ A similar architecture has been recently proposed by Jégou and Seznec [Jégou86].

carried dependences; for other types of loops the polycyclic architecture does not help in finding an optimal solution in polynomial time. The major drawback of the polycyclic approach is the cost of the interconnect.

The polycyclic architecture has been the basis for a commercial machine, the Cydra 5 [Rau89], which has been recently built.

5.1.4 An Optimal Algorithm for the General SLSP Problem with Infinite Resources

The general SLSP problem is NP-complete, however polynomial time algorithms can be found if one is compiling for a machine with unlimited resources (i.e. no resource constraints). One such algorithm, *greedy scheduling*, has been devised by Aiken and Nicolau [Aiken88b]. Greedy scheduling can handle only single cycle operations; in what follows we present a modified version of the algorithm that handles multicyle operations.

Let $\mathcal{L}$ be the input loop, greedy scheduling works by scheduling $\mathcal{L}$'s operations in a greedy fashion, that is as early as dependences permit. This is achieved by unrolling the first $\rho_{\mathcal{L}}$ iterations of $\mathcal{L}$ (for some $\rho_{\mathcal{L}} \in \mathbb{N}$) and applying list scheduling to the unrolled iterations. For a large enough $\rho_{\mathcal{L}}$ the greedy schedule should contain a repeating pattern. The amount of unrolling $\rho_{\mathcal{L}}$, necessary to detect such pattern is determined by the cycles in $\mathcal{L}$'s DDG. Let C be one such data dependence cycle, we define $slope(C) = \delta(C)/\lambda(C)$ (where $\delta(C)$ and $\lambda(C)$ are respectively the sum of the $\delta(e)$ s and $\lambda(e)$ s over all the edges e in C). For each operation $op \in \mathcal{L}$ we define $C[op]$ to be the data dependence cycle of maximum *slope* in which op is involved and $\tau(op) = 2 \cdot \lambda(C[op]) \cdot \delta(C[op]) + 3 \cdot \lambda(C[op])$. Aiken and Nicolau have proved the following:

Theorem: every greedy schedule of a loop $\mathcal{L}$ comprising n iterations, satisfies the following property:

$$\forall op \in \mathcal{L} \quad \forall \tau(op) < i \leq n \quad t_{ii}(i, i + \lambda(C[op])) + \sigma_{i+\lambda(C[op])}(op) - \sigma_i(op) = \delta(C[op])$$

that is after scheduling $\tau(op) + 1 + \lambda(C[op])$ iterations, any occurrence of operation op is scheduled *exactly* $\delta(C[op])$ time steps after the occurrence of op , $\lambda(C[op])$ iterations before.

Thus we just need to set $\rho_{\mathcal{L}} = (\max_{op \in \mathcal{L}} \tau(op)) + 1$. This makes greedy scheduling a polynomial time algorithm, since:

$$\forall op \in \mathcal{L} \quad \delta(C[op]) \leq l \cdot \delta_{max} \quad \text{and} \quad \lambda(C[op]) \leq l \cdot \lambda_{max} \implies \rho_{\mathcal{L}} = O(l^2)$$

(l is the number of operations in $\mathcal{L}$ and δ_{max} , λ_{max} are the maximum values of δ and λ over $\mathcal{L}$'s DDG).

Unfortunately the pattern produced by such algorithm is still too irregular to be practically exploited by a VLIW machine. Consider for instance the loop whose DDG is given in figure 24(a). The greedy schedule obtained for such a loop (assuming single cycle operations) is given in figure 26. The schedules have some undesirable “gaps” that prevent generation of efficient VLIW code (the loop has to be fully unrolled).

To remedy such problem Aiken and Nicolau have introduced the notion of a *region*. Informally, a region $\sigma_i^{-1}(t_1..t_2)$ of σ_i is a time slice of σ_i containing no “gaps”, that is:

$$\sigma_i^{-1}(t_1..t_2) = [\sigma_i^{-1}(t_1); ..; \sigma_i^{-1}(t_2)] \quad \text{where} \quad \sigma_i^{-1}(t) = \{op \in \mathcal{L} \mid \sigma_i(op) = t\}$$

A maximal region is a region which is of maximal size. In the above example σ_5 has two maximal regions: $[\{o7\}; \{o9\}; \{o10\}; \{o12, o13\}; \{o11, o14\}; \{o8\}]$ and $[\{o2\}; \{o1, o4\}; \{o3, o6\}; \{o5\}]$. Note that between any two maximal regions of σ_i there must exist at least one cycle (a “gap”) where no operation of $\mathcal{L}$ is scheduled. These gaps play a key role in generating more concise software pipelined code.

Aiken and Nicolau have presented a condition under which the gaps in a schedule σ_{i+k} can be shrunk to the size of the gaps in schedule σ_i (provided $t_{ii}(i, i+k)$ is increased) without affecting the time optimality of the final code.

iteration schedules						
	σ_1	σ_2	σ_3	σ_4	σ_5	...
0	o1 o2 o7	o7	o7	o7	o7	
1	o3 o4 o8 o9	o9	o9	o9	o9	
2	o5 o6 o10 o11	o2 o10	o10	o10	o10	
3	o12 o13	o1 o4 o12 o13	o12 o13	o12 o13	o12 o13	
4	o14	o6 o11 o14	o11 o14	o11 o14	o11 o14	
5		o3 o8	o8	o8	o8	
6		o5	o2	—	—	
7			o1 o4	—	—	
8			o3 o6	—	—	
9			o5	o2	—	
10				o1 o4	—	
11				o3 o6	—	
12				o5	o2	
13					o1 o4	
14					o3 o6	
15					o5	

Figure 26: Schedules for the loop of figure 24 produced by greedy scheduling. All the initiation intervals are equal to zero ($t_{ii}(i, i+1) = 0$), the above table is therefore identical to the time/iteration diagram.

Theorem: let i_0 and $i_0 + \Delta$ be the first two greedily scheduled iterations of $\mathcal{L}$ such that σ_{i_0} and $\sigma_{i_0+\Delta}$ have the same maximal regions and the inter-region gaps in $\sigma_{i_0+\Delta}$ are as large or larger then in σ_{i_0} . If there is no chain of dependences from a region in σ_{i_0} (except the last) to the last region of $\sigma_{i_0+\Delta}$ then a more “regular” solution (denoted with $'$) to the general SLSP problem with unlimited resources can be obtained from greedy scheduling as follows. Let $t'_{ii} = t_{ii}(i_0, i_0 + \Delta) + \text{len}(\sigma_{i_0+\Delta}) - \text{len}(\sigma_{i_0})$, then:

$$\left\{ \begin{array}{l} \forall 1 \leq i < i_0 + \Delta \quad \left\{ \begin{array}{l} \sigma'_i = \sigma_i \\ t'_{ii}(1, i) = t_{ii}(1, i) \end{array} \right. \\ \forall i_0 + \Delta \leq i \leq n \quad \left\{ \begin{array}{l} \sigma'_i = \sigma_{i_0 + (i - i_0) \bmod \Delta} \\ t'_{ii}(i_0, i) = \lfloor \frac{i - i_0}{\Delta} \rfloor \cdot t'_{ii} + t_{ii}(i_0, i_0 + (i - i_0) \bmod \Delta) \end{array} \right. \end{array} \right.$$

that is we can reduce the inter-region gaps of $\sigma_{i_0+h+k\Delta}$ to the size of the corresponding gaps in σ_{i_0+h} ($0 \leq h < \Delta$, $1 \leq k$) and increase $t_{ii}(i_0, i_0 + h + k \cdot \Delta)$ by the amount the inter-region gaps of $\sigma_{i_0+\Delta}, \sigma_{i_0+\Delta+1}, \dots, \sigma_{i_0+h+k\Delta}$ have been shrunk, without affecting the time optimality of the final result.

Compact code can be generated by considering the schedules and initiation intervals of iterations $i_0, i_0 + 1, \dots, i_0 + \Delta - 1$ as a *single rigid* schedule σ' and packing together (while adjusting array indices) every t'_{ii} th instruction of σ' (iterations 1 through $i_0 - 1$ form the prelude of the software pipeline).

The only problem with this approach is that it no longer guarantees a polynomial time algorithm, in fact: let lcm denote the least common multiple, then $\Delta \geq \text{lcm}_{op \in \mathcal{L}}(\lambda(\mathcal{C}[op]))$, which can potentially be exponential in l (the number of operations in $\mathcal{L}$).

5.2 Complex Software Pipelining Techniques

In this section we survey two pipelining algorithms that are able to process loops with arbitrary conditional jumps. We first review *pipeline scheduling* developed by Ebcioglu [Ebcioglu88,Ebcioglu87] to pipeline sequential-natured code (section 5.2.1). We then survey *perfect pipelining* [Aiken88c], a software pipelining technique based on Nicolau's percolation scheduling (section 5.2.2).

5.2.1 Pipeline Scheduling

Pipeline scheduling [Ebcioglu87] is a software pipelining technique specifically designed for sequential, non-numerical code: loops can have arbitrary bounds and contain if-then-else as well as exit statements, however arrays are assimilated to single indivisible variables. The target machine assumed by pipeline scheduling is an abstract VLIW which extends the PS model (Ebcioglu is currently building such a machine [Ebcioglu88]).

Abstract machine model. Ebcioglu's abstract machine model $\mathcal{N}$ extends Nicolau's PS model (see section 4.4.1), by allowing operations to be performed conditionally, that is the Cj_I part of an instruction I contains operations as well as conditional tests. Formally an instruction I is of the form:

$$L_I : [Op_I, Cj_I, Next_I]$$

where L_I and $Next_I$ are instruction labels, Op_I is a set of register operations¹², and Cj_I is either empty or of the form:

$$\{ \text{IF } c \text{ THEN } I' \text{ ELSE } I'' \}$$

where c is a condition code register and I' and I'' are two unlabeled instructions¹³. Figure 27(a) gives an example. Like for PS instructions can be represented by a binary tree except that in this case every tree edge may be labeled with a set of operations (see figure 27(b)).

The execution of an instruction I takes a *single* cycle, it proceeds as follows. Let T_I be the tree representation of I .

1. Given the current (i.e. old) values of the condition registers, the (unique) execution path in T_I is determined by following the right edge of a conditional jump if the condition is true, and the left branch if the condition is false. For instance, assuming $c1$ and $c2$ are both true, the path selected in the instruction of figure 27(a) would go from L_1 to L_3 .
2. The operations labeling the execution path are executed in parallel by performing all the read first, and then all the writes. If a variable is updated by two different operations the one which is closer to the leaf has priority.
3. The branch taken is given by the label of the leaf of the execution path.

¹² Arrays are assimilated to single indivisible variables, that is operations $a(1):=1$ and $a(2):=2$ are assumed to access the same variable a .

¹³ In what follows instructions such that $Op_I = Cj_I = \phi$ are identified with their $Next_I$ label (for instance $[\phi, \phi, L_2]$ is identified with L_2).

A program for the abstract machine model $\mathcal{N}$ can be represented by a flow graph where each vertex contains a single $\mathcal{N}$ instruction. This representation is similar to the one adopted in for PS, and is likewise termed *parallel program graph*. Like for PS a vertex v of the parallel program graph may have an arbitrary number of outgoing edges according to the structure of the instruction represented by v .

$$\begin{aligned}
 L_1: & \{ \{ a := b + c; b := x(i); i := i + 1; c5 := (i < n) \}, \\
 & \{ \text{IF } c1 \text{ THEN } \{ \{ d := 0 \}, \{ \text{IF (NOT } c2) \text{ THEN } \{ \phi, \phi, L_2 \} \\
 & \quad \text{ELSE } \{ \{ h := y(i); j := j + 1 \}, \phi, L_3 \} \}, \text{ —} \} \\
 & \text{ELSE } \{ \{ d := x(i); b := 0 \}, \phi, L_7 \} \}, \text{ —} \}
 \end{aligned}$$

(a)

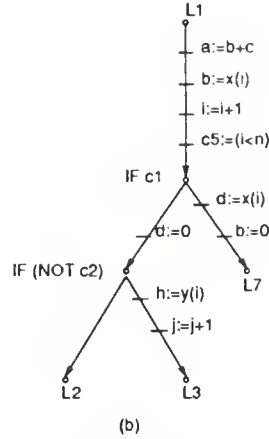


Figure 27: (a) An instruction of Ebcioglu abstract machine model. (b) Tree representation of the instruction.

Pipeline scheduling. Pipeline scheduling is a software pipelining technique explicitly developed for the previous machine model $\mathcal{N}$. The algorithm exploits the ability to execute operations conditionally by performing, as early as possible, instructions of successive iterations on every execution path. Pipeline scheduling has been designed for single entry loops, however in the case of multiple entries an entry instruction is given priority and the algorithm pipelines only those execution paths that originate at the selected entry node. The algorithm is divided in two phases:

phase 1: the input loop is compacted using the greedy-compact transformation of PS, that is operations are pushed up the parallel program graph of the loop body as high up as possible (see section 4.4.2)¹⁴.

phase 2: Let $\mathcal{L}$ be the resulting loop and let $I_1, I_2, \dots, I_l$ be the VLIW instructions composing $\mathcal{L}$ (i.e. the nodes of the compacted parallel program graph). Let I_1 be the selected entry instruction of $\mathcal{L}$ and let i be any arbitrary iteration, then instruction $I_1(i + 1)$ (the first instruction of iteration $i + 1$) is

¹⁴Actually any global compaction transformation is acceptable for phase 1.

moved up all the execution paths starting at $I_1(i)$ and ending at $I_1(i+1)$, as much as possible. This is accomplished by trying to schedule $I_1(i+1)$ concurrently with each one of the branches of the instructions directly reachable from $I_1(i)$. Due to resource and/or data dependence constraints it may not be possible to pack $I_1(i+1)$ with some of the branches of the direct successors of $I_1(i)$. In this case we try to schedule $I_1(i+1)$ concurrently with each one of the branches of the instructions that are pointed by the branches for which the initial concurrent scheduling failed. This last step is repeated until $I_1(i+1)$ is scheduled on each execution path from $I_1(i)$ to $I_1(i+1)$ as high up as resource and data dependence constraints permit (note that on some execution paths it may not be possible to schedule $I_1(i+1)$ at all). After $I_1(i+1)$ has been moved up, the children (i.e. direct successors) of $I_1(i+1)$ are pushed up in the same manner. However if one of these children, say $I_r(i+1)$, was such that $I_1(i+1)$ has been scheduled concurrently with one or more of $I_r(i)$'s branches, then we move up $I_r(i+1)$ with $I_1(i+2)$ attached at the corresponding branches, instead of just moving $I_r(i+1)$ alone.

Given that the machine model $\mathcal{N}$ allows only register operations, all loop carried dependences go from iteration to the next (i.e. if for some edge e of $\mathcal{L}$'s DDG, $\lambda(e) > 0$ then $\lambda(e) = 1$). This is what allows $I_1(i+2)$ to be moved up simultaneously with $I_r(i+1)$ and still guarantee that $I_r(i+1)$ will be pushed up all the execution paths of iteration i as much as possible, because there are no dependences between operations in iteration i and $I_1(i+2)$ ¹⁵. The ability to move $I_1(i+2)$ up concurrently with $I_r(i+1)$ not only saves work but also, and especially, guarantees that a repetitive pattern will be generated (that is we can obtain a compact pipelined loop schedule).

After all the children of $I_1(i+1)$ have been moved up all the execution paths of iteration i as much as possible, we focus on the grand children of $I_1(i+1)$ (i.e. the direct successors of the direct successors of $I_1(i+1)$). Let $I_q(i+1)$ be one such grand children, $I_q(i+1)$ is pushed up all the execution paths of iteration $i+1$ as much as possible. However if a children of $I_1(i+1)$, say $I_p(i+1)$, has been scheduled concurrently with one or more of $I_q(i)$'s branches, then we push up $I_q(i+1)$ with $I_p(i+2)$ attached at the corresponding branches. Note that if $I_1(i+1)$ has been scheduled concurrently with one or more of $I_p(i)$ branches then we push up $I_q(i+1)$, $I_p(i+2)$, $I_1(i+3)$, where $I_p(i+2)$ and $I_1(i+3)$ are attached to the corresponding branches of (respectively) $I_q(i+1)$ and $I_p(i+2)$.

After all the grand children of $I_1(i+1)$ have been moved up, we move up the children of the grand children of $I_1(i+1)$, and so on. This process is repeated until all the instructions of iteration $i+1$ have been moved up iteration i as much as possible (note that by moving up instructions of iteration $i+1$ we simultaneously pull up instructions of iterations $i+2$, $i+3$, ... and effectively pipeline iterations i , $i+1$, $i+2$, $i+3$, ...). Note also that arbitrary execution gaps may happen in iterations $i+1$, $i+2$, ... while they are concurrently executing with iteration i (flexible pipeline)¹⁶.

A more detailed description of the algorithm follows. Let $\mathcal{F}$ be the parallel program graph of the initial compacted loop, and let $\mathcal{F}^*$ be the parallel program graph of the software pipelined loop. Pipeline scheduling proceeds as follows.

¹⁵Note that a loop $\mathcal{L}_a$ with arbitrary carried dependences can be converted into one where carried dependences go from one iteration to the next, by unrolling $\mathcal{L}_a$ ($\max_{e \in \text{DDG of } \mathcal{L}_a} \lambda(e)$) times.

¹⁶While the previous software pipelining algorithms produce *rigid* pipelines (i.e. the execution time of each iteration is predictable at compile time), pipeline scheduling creates *flexible* pipelines (i.e. the execution of an iteration may contain variable time gaps not predictable at compile time).

1. Copy n_1 , the selected entry node of $\mathcal{F}$ to $\mathcal{F}'$ and name it $[n_1]$.
2. For each direct successor n_c of n_1 , different from n_1 and belonging to $\mathcal{F}$ (i.e. it is not an exit node), enqueue $(n_c, [n_1])$ in a queue Q if not already there.
3. Let $(n_p, [n_{j_k}, \dots, n_{j_1}])$ be the element at the front of Q , where n_p is a node of $\mathcal{F}$ and $[n_{j_k}, \dots, n_{j_1}]$ a node of $\mathcal{F}'$ ¹⁷. Copy n_p from $\mathcal{F}$ to $\mathcal{F}'$, name the new node $[n_p, n_{j_k}, \dots, n_{j_1}]$, then

for each branch b of n_p which does not branch out of the loop do

(a) b branches back to n_1

if $[n_{j_k}, \dots, n_{j_1}]$ can execute concurrently with branch b

(i.e. resource and data dependence constraints are not violated)¹⁸ then

append a copy of $[n_{j_k}, \dots, n_{j_1}]$ to the end of b

else

set b to branch to $[n_{j_k}, \dots, n_{j_1}]$

(b) b branches to $n_r \neq n_1$

if $[n_{j_k}, \dots, n_{j_1}]$ can execute concurrently with branch b then

make a copy of $[n_{j_k}, \dots, n_{j_1}]$, change the end of each of its branches

to point from $[n_{e_w}, \dots, n_{e_1}]$ to $[n_r, n_{e_w}, \dots, n_{e_1}]$ and

append the modified $[n_{j_k}, \dots, n_{j_1}]$ to the end of b .

Enqueue $(n_r, [n_{e_w}, \dots, n_{e_1}])$ if not already in Q or in $\mathcal{F}'$

else

set b to branch to $[n_r, n_{j_k}, \dots, n_{j_1}]$ and

enqueue $(n_r, [n_{j_k}, \dots, n_{j_1}])$ if not already in Q or $\mathcal{F}'$

4. Dequeue $(n_p, [n_{j_k}, \dots, n_{j_1}])$ and repeat steps 3 and 4 until Q is empty.

The prelude and postlude of the pipelined loop are automatically generated by the algorithm, if necessary.

Ebcioğlu has proven [Ebcioğlu87] that the above algorithm terminates, and that the resulting parallel program graph $\mathcal{F}'$ is semantically equivalent to the input parallel program graph $\mathcal{F}$. Pipeline scheduling has a potential exponential code increase, in practice however, code explosion would be prevented by resource shortages.

We demonstrate pipeline scheduling on an example (taken from [Ebcioğlu87]). Consider the C program of figure 28(a). The code copies a string a composed of upper and lower case letters, to a string b . In the copying process lower case letters are converted to upper case, and upper case letters are preceded by a backslash. This is the transformation that UNIX performs when communicating with upper case only terminals. The

¹⁷ We are about to generate node $[n_p, n_{j_k}, \dots, n_{j_1}]$ of $\mathcal{F}'$ by trying to append the instruction in $[n_{j_k}, \dots, n_{j_1}]$ to each of the branches of the instruction contained in n_p (in this way the current and the k future iterations $i, i+1, i+2, \dots, i+k$ will advance by one instruction, respectively $n_p, n_{j_k}, n_{j_{k-1}}, \dots, n_{j_1}$). If there is a branch b to which appending is not possible (due to data dependence or resource conflicts) then the computation along branch b of instruction n_p of iteration i will execute alone in $\mathcal{F}'$, that is the current iteration i will advance by one instruction (n_p) while the k future iterations will wait.

¹⁸ When $[n_{j_k}, \dots, n_{j_1}]$ updates a variable v whose old value is used in an instruction reachable from branch b , $[n_{j_k}, \dots, n_{j_1}]$ cannot be appended to b . On the fly renaming of v allows $[n_{j_k}, \dots, n_{j_1}]$ to be appended to b , and is performed whenever possible by pipeline scheduling.

intermediate code for the program is given in figure 28(b). The code obtained after compacting the program with PS (and performing on the fly renaming for c) is given in 28(c). The tree representation of instructions L_1 , L_2 , L_3 and L_4 is given in figure 29. Figure 30 demonstrates pipeline scheduling on the loop of figure 28. The instructions generated are listed in the order in which they were added to $\mathcal{F}$. The steady state of the pipelined loop executes at 1-2 cycles per iteration, depending on whether the character copied during the current iteration is lower or upper case. The loops produced by pipeline scheduling executes operations as soon as possible, regardless of whether the iterations to which such operations belong will later have to wait because of dependence conflicts (flexible pipeline). Let us illustrate this point by examining the behavior of the pipelined loop on the input $a = \text{"AbC"}$. The output will be $b = \text{"\AB\C"}$. The following table shows the evolution of the software pipeline: the leftmost column gives the cycle in which operations are executed, the next six columns tabulate the operations that are executed by each iteration. Iterations 4 through 6 do useless work (we assume that it is safe to access a past its end). The rightmost column gives the instruction actually executed by the pipelined loop. As one can see from the table iterations 2 and 3 wait while iteration 1 completes.

cycle	iteration						instruction executed
	1	2	3	4	5	6	
1	L_1						$[L_1]$
2	L_2	L_1					$[L_2, L_1]$
3	L_3	L_2	L_1				$[L_3, L_2, L_1]$
4	L_4	—	—				$[L_4, L_3, L_2, L_1]$
5		L_3	L_2	L_1			$[L_3, L_2, L_1]$
6			L_3	L_2	L_1		$[L_3, L_2, L_1]$
7			L_4	—	—		$[L_4, L_3, L_2, L_1]$
8				L_3	L_2	L_1	$[L_3, L_2, L_1]$
9							E_1

Hardware implementation of the abstract machine. Pipeline scheduling heavily depends on the ability to perform several conditional computations simultaneously. Ebcioglu is currently designing a machine with such capabilities [Ebcioglu88] (figure 31 gives the block diagram of the machine data path). The machine is programmed via an 1800 bits long instruction, and has N_{ALU} (16) 32-bits combinatorial¹⁹ ALUs (numbers in parenthesis give the values for the current prototype) capable of executing integer as well as floating point operations. The results of comparison operations are sent to 2-bits condition code registers (one per ALU), while other operations send their results to a multiported register file containing N_{reg} (128) 33-bits registers (the multi-ported register file has N_{ALU} input ports connected to the N_{ALU} ALU outputs and $2N_{ALU}$ output ports connected to the $2N_{ALU}$ ALU inputs). Operations can be interruptible and non-interruptible. Non-interruptible operations are used when unsafe code motions are performed (for instance when $b := c/a$ is moved past if ($a \neq 0.0$) then ...). If an uninterruptible operation yields an invalid result, the invalid bit of the destination register (2nd bit for a condition code register and 33rd bit for the other registers) is set and no exception raised. If one of the arguments of an interruptible operation is invalid an exception occurs.

¹⁹ This choice is motivated by the faster execution time of combinatorial over pipelined hardware. Pipelined hardware trades fast execution time over the ability to start operations every cycle. In inherently sequential code an iteration i will often have to wait for a result from iteration $i - 1$. It is therefore more important that the overall execution time of operations be small rather than being able to start operations every cycle.

```

char a[BUFSIZE], b[BUFSIZE];
int c,s1,s2;

s1=s2=0;

while(c=a[s1++]) {
    if(c>='A' && c<='Z') {
        b[s2++]='\';
        b[s2++]=c;
    }
    else {
        b[s2++]=c+('A'-'a');
    }
}

/* b and s2 are live here */

```

(a)

```

Loop:
    c := a(s1)
    s1 := s1+1
    cc1 := (c=0)
    IF cc1 GOTO Exit
    cc2 := (c<'A')
    IF cc2 GOTO Lower_case
    cc3 := (c>'Z')
    IF cc3 GOTO Lower_case
    b(s2) := '\'
    s2 := s2+1
    b(s2) := c
    s2 := s2+1
    GOTO Loop
Lower_Case:
    t := c+('A'-'a')
    b(s2) := t
    s2 := s2+1
    GOTO Loop
Exit:

```

(b)

```

L1: [{ c:=a(s1); s1:=s1+1},φ,L2]
L2: [{ cc1:=(c=0); cc2:=(c<'A'); c3:=(c>'Z'); t:=c+('A'-'a'); c':=c},φ,L3]
L3: [φ,{IF cc1 THEN E1
      ELSE [{b(s2):=t; s2:=s2+1}, {IF cc2 THEN L1
      ELSE [φ,{IF cc3 THEN L1
      ELSE [{b(s2):='\' ; c'':=c'}, φ,L4] },-] },-] },-]
L4: [{b(s2):=c''; s2:=s2+1},φ,L1]
E1:

```

(c)

Figure 28: (a) A simple C program. (b) Its intermediate code representation. (c) The program compacted with PS.

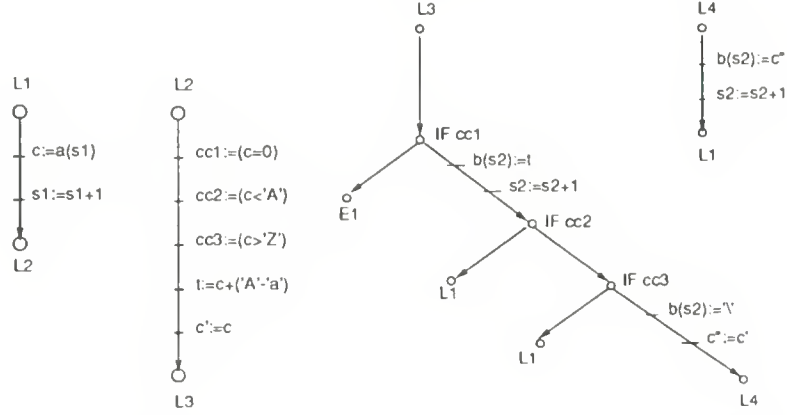


Figure 29: Tree representation of instructions L_1 , L_2 , L_3 and L_4 .

Memory loads as well have non-interruptible version, this is to prevent traps when protected or non-existent memory locations are accessed (in the previous example for instance, we want to prevent exceptions from occurring when accessing string a past its end).

Data memory is partitioned in N_{banks} (8) memory banks accessed through a common memory controller with $N_{ALU}/2$ ports. The left input of ALU 0 and the output of ALU 1 constitute the address and data for memory port 0, the left input of ALU 2 and the output of ALU 3 constitute the address and data for memory port 1, etc. (see figure 31). Memory access takes a single cycle unless there is a bank conflict. When such a conflict occurs the instruction clock freezes until all the memory requests are satisfied in increasing port order. Such hardware assist has been provided because compile time memory bank disambiguation is very difficult in the case of non-numerical code.

To execute an instruction of the abstract machine $\mathcal{N}$ the real machine needs to determine the correct branching address and perform the computations on the selected execution path. This is achieved as follows.

Determining the branching address. In the real machine an instruction is allowed to branch only to N_t (4) target addresses. Each of the N_t target addresses is associated with a true and false condition mask (t-mask and f-mask) which specify a subset of the condition code registers. The branching address is the target address such that all the condition code registers specified by the t-mask (f-mask) are set to true (false).

Performing computations on the selected execution path. Determining the branching address takes practically a whole cycle. As instructions must execute in a single cycle, *all* the computations specified in an instruction tree are executed while the branching address is determined. The register stores however, are executed conditionally. A transfer enable mask (te-mask) is attached to each operation's destination register. The te-mask has N_t bits, one for each target address. Assume that the branching address has been determined to be the i th target address. Before storing the result in a destination register the machine checks whether the i th bit of its te-mask is 1. If the i th bit is zero the store is

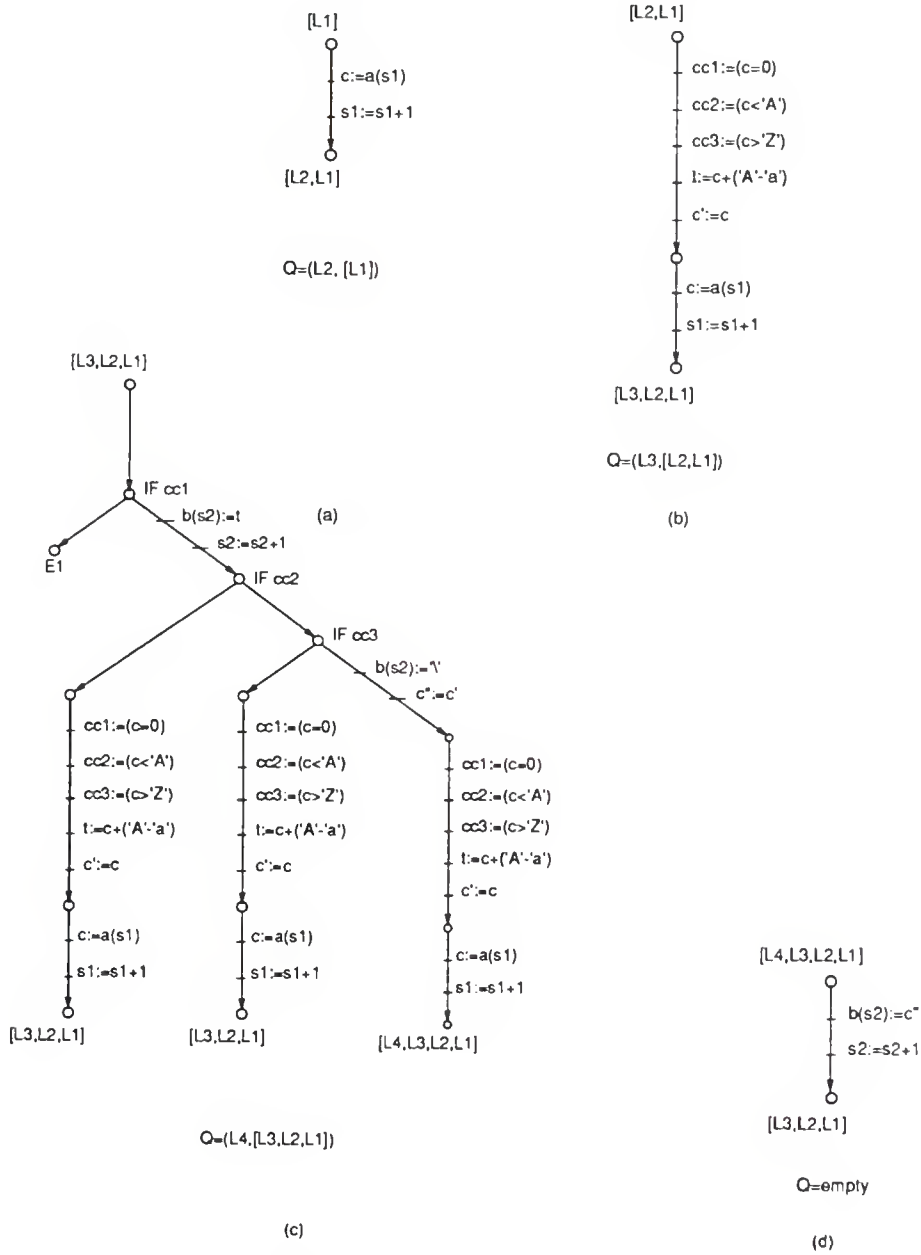


Figure 30: (a) $[L_1]$ is generated. (b) $[L_2, L_1]$ is generated. (c) $[L_3, L_2, L_1]$ is generated. (d) $[L_4, L_3, L_2, L_1]$ is generated. $[L_3, L_2, L_1]$ cannot execute concurrently with L_4 , because of data dependence conflicts between s_2 and $b(s_2)$.

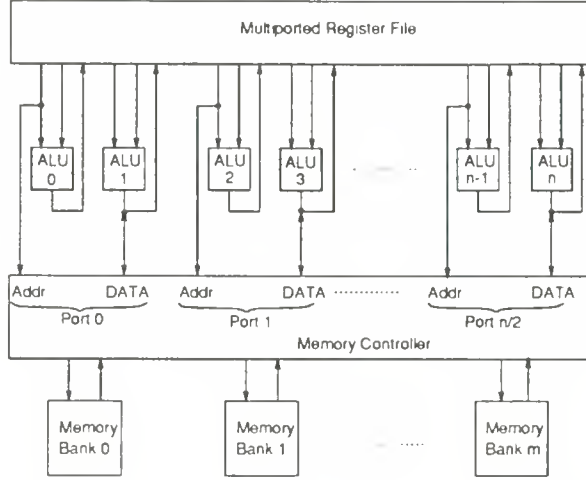


Figure 31: Block diagram of the machine data path. In the picture n stands for N_{ALU} , and m for N_{banks} .

not performed. At the end of the instruction cycle those multi-cycle operations (such as floating point computations) which are computing useless results are aborted.

5.2.2 Perfect Pipelining

Aiken and Nicolau [Aiken88c] have presented a technique, *perfect pipelining*, which achieves the same results as unbounded loop unrolling followed by global compaction. The idea behind perfect pipelining is to move entire iterations up without creating execution gaps. This generates rigid pipelines as opposed to the flexible pipelines created by pipeline scheduling. However while simple pipeline scheduling imposes a minimum of 1 cycle delay between the execution of successive iterations, perfect pipelining can start consecutive iterations simultaneously if resource and data dependence constraints permit (loop unrolling followed by pipeline scheduling can achieve a similar result). Perfect pipelining proceeds as follows. Let $\mathcal{L}$ be a single entry loop.

1. Unroll $\mathcal{L}$ say k times (k will be determined later). Call the unwound loop $\mathcal{L}^k$.
2. Move iteration $i + 1$ as far up in the parallel program graph of $\mathcal{L}^k$ on as many paths as possible by using the core transformations of PS. Iteration $i + 1$ is shifted in a rigid fashion, that is no execution gaps are allowed in iteration $i + 1$ (i.e. between the first and last operations of iteration $i + 1$ there must be an operation of iteration $i + 1$ scheduled on every cycle). As iteration $i + 1$ moves through the parallel program graph of $\mathcal{L}^k$ copies of operations are generated where paths split, effectively creating other copies of iteration $i + 1$. These copies are shifted up as well.
3. When the shift up process stops because none of the copies of iteration $i + 1$ can any longer move (due to resource or data dependence constraints), the move-up process is repeated for iteration $i + 2, i + 3, \dots, i + k$. Call the final result $\mathcal{L}_{move-up}^k$.

4. Consider each execution path P starting at the root of the parallel program graph of $\mathcal{L}_{move-up}^k$. Let v_P be a vertex of P containing at least one operation from iteration $i + 1$. Then for a large enough k there must exist a node v'_P of P which has the same structure as v_P . Replace all the edges (u, v'_P) in $\mathcal{L}_{move-up}^k$ parallel program graph with edges (u, v_P) and increment the loop index along those edges by the appropriate amount.
5. Repeat step 4 until all execution paths and all vertices containing at least one operation from iteration $i + 1$ have been explored, then delete all unreachable nodes of $\mathcal{L}_{move-up}^k$ parallel program graph. This is the final result of perfect pipelining.
6. The value of k can be determined on the fly. Initially set $k = 2$. If after applying steps 1 through 5 there still exists some vertex v_P (as defined in 4) on some path P for which no node v'_P (of P) having the same structure can be found, then we increment k and add iteration $i + k$ to the end of $\mathcal{L}_{move-up}^k$, and repeat the whole process till convergence (Aiken and Nicolau have proved that convergence will sooner or later arise).

The algorithm has an exponential worst case running time which, according to Aiken and Nicolau, never arises in practice. As we previously said perfect pipelining produces a rigid schedule i.e., once the execution of some iteration starts it will run to completion without pausing.

We illustrate perfect pipelining on the following loop:

```

FOR i IN 1..n LOOP
  IF t_key(i)=key THEN
    s:=s+t_val(i);
  END IF;
END LOOP;

```

The parallel program graph of the above loop is given in figure 32(a). T denotes test IF $t(i)=key$ and S statement $s:=s+t\_val(i)$. Loop control code has been omitted for simplicity, i is assumed to be incremented on the loop back edges. For simplicity we assume that the test IF $t\_key(i)=key$ does the comparison and the branching at the same time (this is not allowed by the PS model but allows to obtain a final parallel program graph of manageable size). Figure 32(b) shows the loop unrolled 4 times. We assume that our target machine can only perform two conditional tests per instruction. After pushing up iterations $i + 1$ and $i + 2$ as much as possible we obtain the parallel program graphs of figure 32(c) and 32(d). Tests within a node are arranged as shown next to the root node T0T1. Once all the iterations have been pushed up we obtain the parallel program graph of 32(e). After applying step 4 to the nodes indicated by the dotted arcs we obtain the parallel program graph of 32(f). The final program graph produced by perfect pipelining is obtained by applying step 4 to the nodes of 32(f) having similar structure. The final result is given in 32(g).

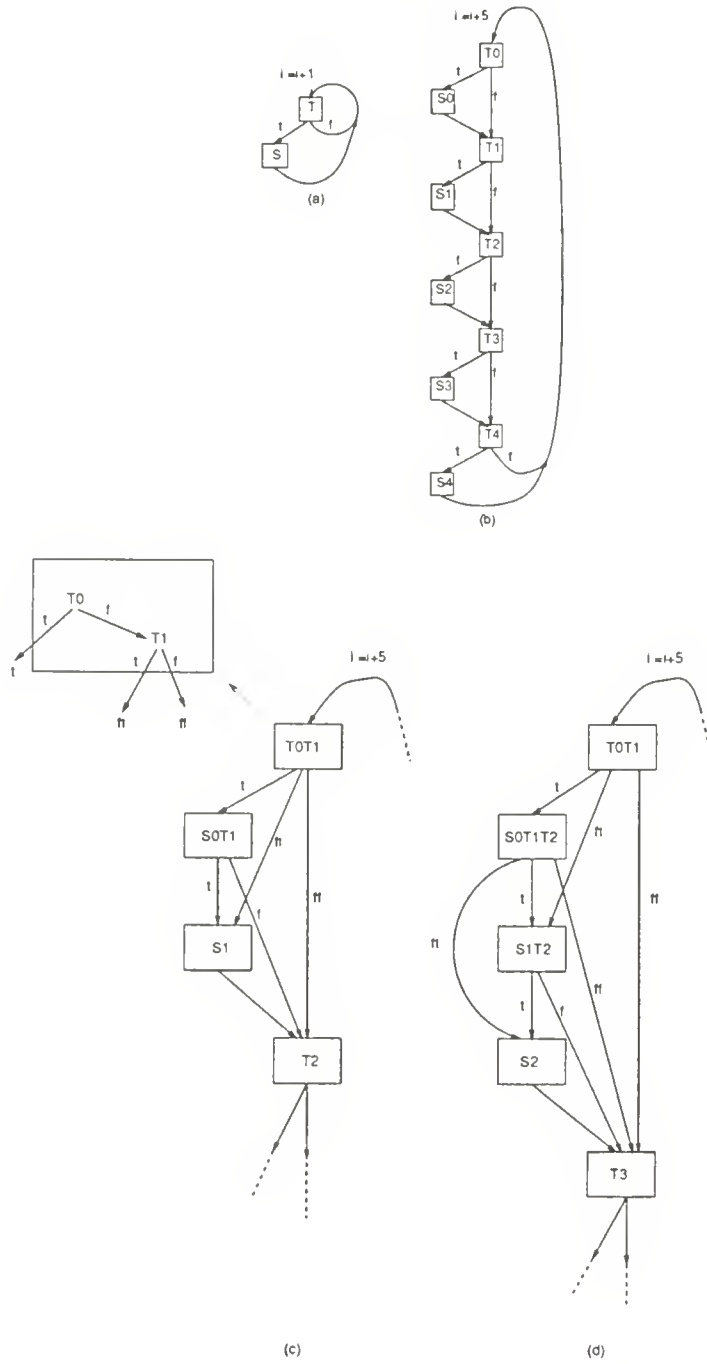
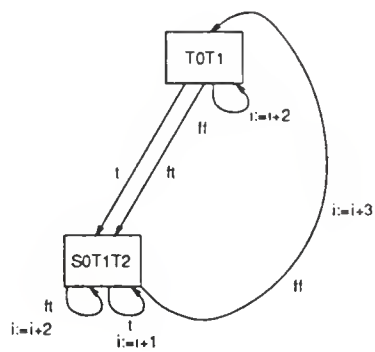
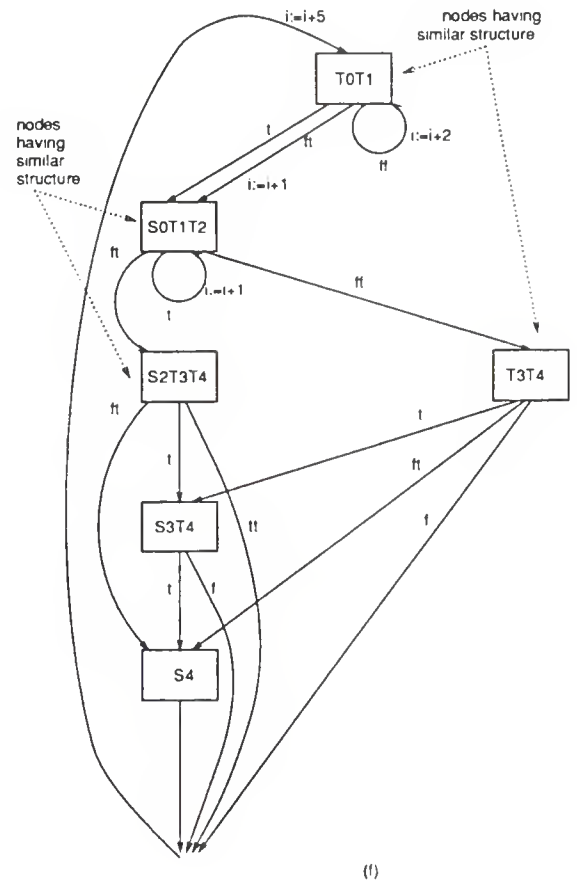
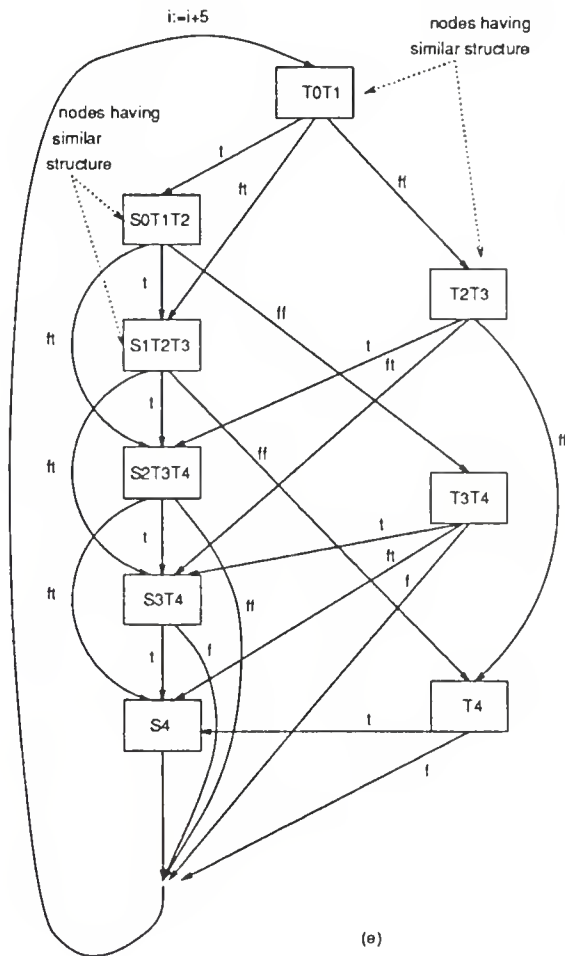


Figure 32: (a) Initial parallel program graph. (b) The loop unrolled 4 times. (c) Iteration $i + 1$ has been pushed up. (d) Iteration $i + 2$ has been pushed up. (e) All 4 iterations have been pushed up. (f) Step 4 has been applied to the nodes pointed by the dotted arcs. (g) Final parallel program graph.



6 Conclusion

VLIWs are able to achieve high performance at low cost thanks to the hardware/compiler symbiosis which allows relatively inexpensive hardware to be employed.

Two types of compilation techniques have been surveyed: global compaction and software pipelining. Both techniques are needed for generation of efficient code. As the ratio of value-returning operations to conditional jumps is more or less fixed in a given algorithm (10% to 30%), VLIWs have to incorporate some kind of multi-branching capabilities: three such multi-branching mechanisms have been surveyed.

Given the interesting results a natural question to ask is whether the VLIW concept scales up. Cycles in program data dependence graphs are serious performance bottlenecks, as the slope $\delta(C)/\lambda(C)$ of a cycle C imposes an upper bound on program speedup. This suggests that there is a limit to the scalability of VLIWs.

Programs contain three kinds of parallelism: fine grain, coarse grain and task level. When all fine grain parallelism has been uncovered, higher levels of parallelism have to be extracted if greater speedups are desired. VLIWs are ill suited to exploit coarse and task level parallelism. Parallel processors, however have been designed to exploit such types of parallelism, but have a too high overhead to use fine grain parallelism effectively. An hybrid scheme combining multiprocessor and VLIW architectures could effectively use all types of parallelism. A compiler would automatically extract fine grain as well as coarse grain parallelism, while the user would be responsible for task level parallelism. The program dependence graph would be a valuable tool throughout the compilation process as it allows simultaneous extraction of fine grain as well as coarse grain parallelism.

References

- [Aho86] Aho A., Sethi R., and Ullman A. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
- [Aiken88a] Aiken A. and Nicolau A. "A development environment for horizontal microcode". *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.
- [Aiken88b] Aiken A. and Nicolau A. "Optimal loop parallelization". In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 308–317, ACM, 1988.
- [Aiken88c] Aiken A. and Nicolau A. "Perfect pipelining: a new loop parallelization technique". In *European Symposium on Programming*, pages 221–235, Springer-Verlag, Lecture notes in Computer Science No. 300, June 1988.
- [Allen87] Allen R. and Kennedy K. "Automatic translation of Fortran programs to vector form". *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
- [Annaratone87] Annaratone M., Arnould E., Gross T., Kung H. T., Lam M., Menzilioğlu O., and Webb J. A. "The WARP computer: architecture, implementation, and performance". *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.
- [Arvind80] Arvind. "Decomposing a program for multiple processor systems". In *International Conference on Parallel Processing*, pages 7–16, IEEE, August 1980.
- [Burke88] Burke M., Cytron R., Ferrante J., Hsieh W., Sarkar V., and Shields D. "Automatic discovery of parallelism: a tool and an experiment". In *ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages*, pages 77–84, ACM, July 1988.
- [Carnevali86] Carnevali P., Vitaletti M., and Zecca V. "Microtasking on IBM multiprocessors and its implications on automatic parallelization". In *Parallel Algorithms and Architectures*, pages 219–228, Elsevier Science Publishers B. V. (North-Holland), 1986.
- [Charlesworth81] Charlesworth A. E. "An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family". *IEEE Computer*, 14(9):18–27, September 1981.
- [Coffman76] Coffman E. G. *Computer and Job-shop Scheduling Theory*. John Wiley and Sons, 1976.
- [Colwell88] Colwell R. P., Nix R. P., O'Donnel J. J., Papworth D. B., and Rodman P. K. "A VLIW architecture for a trace scheduling compiler". *IEEE Transactions on Computers*, C-37(8):967–979, August 1988.
- [Cytron86] Cytron R. "Doacross: beyond vectorization for multiprocessors". In *International Conference on Parallel Processing*, pages 836–844, IEEE, August 1986.
- [Ebcioglu87] Ebcioglu K. "A compilation technique for software pipelining of loops with conditional jumps". In *20th Annual Microprogramming Workshop*, pages 69–79, IEEE and ACM, December 1987.

- [Ebcioglu88] Ebcioglu K. "Some design ideas for a VLIW architecture for sequential-natured software". In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, pages 1–21, North-Holland, April 1988.
- [Ellis86] Ellis J. R. *Bulldog: A Compiler for VLIW Architectures*. ACM doctoral dissertation awards; 1985, MIT Press, 1986.
- [Ferrante87] Ferrante J., Ottenstein K. J., and Warren J. D. "The program dependence graph and its use in optimization". *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [Fisher79] Fisher J. A. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources*. PhD thesis, New York University, October 1979.
- [Fisher81a] Fisher J. A. "Trace scheduling: a technique for global microcode compaction". *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.
- [Fisher81b] Fisher J. A., Landskov D., and Shriver B. D. "Microcode compaction: looking backward and looking forward". In *National Computer Conference*, pages 95–102, AFIP Press, July 1981.
- [Fisher83] Fisher J. A. "Very long instruction word architectures and the ELL-512". In *10th Annual Symposium on Computer Architecture*, pages 140–150, IEEE and ACM, June 1983.
- [Fisher84a] Fisher J. A. "The VLIW machine: a multiprocessor for compiling scientific code". *IEEE Computer*, 17(7):45–53, July 1984.
- [Fisher84b] Fisher J. A., Ellis J. R., Ruttenberg J. C., and Nicolau A. "Parallel processing: a smart compiler and a dumb machine". In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, pages 37–47, ACM, June 1984.
- [Fisher84c] Fisher J. A. and O'Donnel J. J. "VLIW machines: multiprocessors we can actually program". In *Compcon*, pages 299–305, IEEE and ACM, June 1984.
- [Foster72] Foster C. C. and Riseman E. M. "Percolation of code to enhance parallel dispatching and execution". *IEEE Transactions on Computers*, C-21(12):1411–1415, December 1972.
- [Garey79] Garey M. R. and Johnson D. S. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman, 1979.
- [Gupta87a] Gupta R. *A reconfigurable LIW architecture and its compiler*. PhD thesis, University of Pittsburgh, Pittsburgh, PA, 1987.
- [Gupta87b] Gupta R. and Soffa M. L. "A reconfigurable LIW architecture". In *Proceedings of the International Conference on Parallel Processing*, pages 893–900, IEEE, August 1987.
- [Gupta87c] Gupta R. and Soffa M. L. "Region scheduling". In *Proceedings of the 2nd International Conference on Supercomputing*, pages 141–148, May 1987.

- [Gupta88] Gupta R. and Soffa M. L. "Detecting and redistributing parallelism in program regions". 1988. Submitted to *IEEE Transactions on Software Engineering*.
- [Jégou86] Jégou Y. and Seznec A. "Data synchronized pipeline architecture: pipelining in multi-processor environments". *Journal of Parallel and Distributed Computing*, 3(4):508–526, December 1986.
- [Karplus85] Karplus K. and Nicolau A. "Efficient hardware for multi-way jumps and pre-fetches". In *18th Annual Microprogramming Workshop*, pages 11–18, IEEE and ACM, December 1985.
- [Karplus86] Karplus K. and Nicolau A. "A compiler-driven supercomputer". *Applied Mathematics and Computations*, 20(2):95–110, September 1986.
- [Kogge77] Kogge P. "The microprogramming of pipelined processors". In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 63–69, 1977.
- [Kuck81] Kuck D. J., Kuhn R. II., Leasure B., and Wolfe M. "Dependence graphs and compiler optimizations". In *8th ACM Symposium on Principles of Programming Languages*, pages 207–218, ACM, 1981.
- [Lah83] Lah J. and Atkins D. E. "Tree compaction of microprograms". In *16th Annual Microprogramming Workshop*, pages 23–33, IEEE and ACM, October 1983.
- [Lam87] Lam M. *A systolic array optimizer*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1987.
- [Lam88] Lam M. "Software pipelining: an effective scheduling technique for VLIW machines". In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, ACM, June 1988.
- [Landskov80] Landskov D., Davidson S., Shriver B., and Mallet P. W. "Local microcode compaction techniques". *ACM Computing Surveys*, 12(3):261–294, September 1980.
- [Linn83] Linn J. L. "SRDAG compaction – a generalization of trace scheduling to increase the use of global context information". In *16th Annual Microprogramming Workshop*, pages 11–22, IEEE and ACM, October 1983.
- [Mueller86] Mueller R. A., Su B., Duda M. R., and Plomondon B. L. "A case study in signal processing microprogramming using the URPR software pipelining technique". In *19th Annual Microprogramming Workshop*, 1986.
- [Nicolau81] Nicolau A. and Fisher J. A. "Using an oracle to measure potential parallelism in single instruction stream programs". In *14th Annual Microprogramming Workshop*, pages 171–182, IEEE and ACM, October 1981.
- [Nicolau84a] Nicolau A. *Parallelism, Memory Anti-aliasing and Correctness Issues for a Trace Scheduling Compiler*. PhD thesis, Yale University, New Have, CT, June 1984.
- [Nicolau84b] Nicolau A. and Fisher J. A. "Measuring the parallelism available for very long instruction word architectures". *IEEE Transactions on Computers*, C-33(11):968–976, November 1984.

- [Nicolau85a] Nicolau A. *Percolation scheduling: A parallel compilation technique*. Technical Report TR-85-678, Cornell University, Department of Computer Science, Ithaca, NY, May 1985.
- [Nicolau85b] Nicolau A. "Uniform parallelism exploitation in ordinary programs". In *International Conference on Parallel Processing*, pages 614–618, IEEE, August 1985.
- [Padua86] Padua D. A. and Wolfe M. J. "Advanced compiler optimizations for supercomputers". *Communications of the ACM*, 29(12):1184–1201, December 1986.
- [Rau81] Rau B. R. and Glaeser C. D. "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing". In *14th Annual Microprogramming Workshop*, pages 183–198, IEEE and ACM, October 1981.
- [Rau82a] Rau B. R., Glaeser C. D., and Picard R. L. "Efficient code generation for horizontal architectures: compiler techniques and architectural support". In *9th Annual Symposium of Computer Architecture*, pages 131–139, IEEE and ACM, April 1982.
- [Rau82b] Rau B. R., Picard R. L., Glaeser C. D., and Greenawalt E. M. "The polycyclic architecture: a statically scheduled dataflow architecture". *IEEE Computer*, 15(12), December 1982.
- [Rau89] Rau B. R., Yen D. W., W Y., and Towle R. A. "The Cydra 5 departmental supercomputer". *IEEE Computer*, 22(1):12–35, January 1989.
- [Riseman72] Riseman E. M. and Foster C. C. "The inhibition of potential parallelism by conditional jumps". *IEEE Transactions on Computers*, C-21(12):1405–1411, December 1972.
- [Sharir81] Sharir M. "A strong connectivity algorithm and its application in data flow analysis". *Computers and Mathematics with Applications*, 7(1):67–72, 1981.
- [Su84] Su B., Ding S., and Jin L. "An improvement of trace scheduling for global microcode compaction". In *17th Annual Microprogramming Workshop*, pages 78–85, IEEE and ACM, December 1984.
- [Su85] Su B. and Ding S. "Some experiments in global microcode compaction". In *18th Annual Microprogramming Workshop*, pages 175–180, IEEE and ACM, December 1985.
- [Su86] Su B., Ding S., and Xia J. "URPR – an extension of URCR for software pipelining". In *19th Annual Microprogramming Workshop*, 1986.
- [Su87] Su B., Ding S., Wang J., and Xia J. "GURPR – a method for global software pipelining". In *20th Annual Microprogramming Workshop*, 1987.
- [Tarjan72] Tarjan R. E. "Depth first search and linear graph algorithms". *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [Tjaden70] Tjaden G. S. and Flynn M. J. "Detection and parallel execution of independent instructions". *IEEE Transactions on Computers*, C-19(10):889–895, October 1970.
- [Tokoro78] Tokoro M., Takizuka T., E T., and I Y. "A technique of global optimization of microprograms". In *11th Annual Workshop on Microprogramming*, pages 41–50, ACM and IEEE, 1978.

- [Touzeau84] Touzeau R. F. “A fortran compiler for the FPS scientific computer”. In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, pages 48–57, ACM, June 1984.
- [Veen86] Veen A. H. “Dataflow machine architecture”. *Computing Surveys*, 18(4):365–396, December 1986.

NYU COMPSCI TR-435
Gasperoni, Franco
Compilation techniques for
VLIW architectures.

DATE DUE | BORROWER'S NAME

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.

[illegible]

